Fundamentos de sistemas operativos: una aproximación práctica usando Linux

Enrique Soriano-Salvador, Gorka Guardiola Múzquiz
GSyC, Universidad Rey Juan Carlos

2 de septiembre de 2022
1.01
Para obtener la versión más reciente de este documento:
https://honecomp.github.io

Índice de figuras

Índice general

Lista de Programas

0 Sobre este libro

Este es un libro de sistemas operativos. Lo empezamos a escribir durante el primer confinamiento causado por la pandemia COVID-19, como parte del material online para dar soporte a nuestros alumnos. Como tal, es un libro eminentemente práctico, aunque hay capítulos que incluyen conceptos más teóricos. Esos capítulos se pueden complementar con otras referencias bibliográficas clásicas (ver por ejemplo [1, 2, 3]). Los ejemplos se han probado en Linux (específicamente, en una distribución Ubuntu), pero en principio deberían ser fáciles de trasladar a cualquier sistema tipo Unix.

0.0.1. Cómo usar este libro

Este libro puede cubrir parte de una asignatura introductoria de uso de un sistema de tipo Unix, usando los capítulos 1, 2 y parte de 4 (dejando fuera administración y otros temas de desarrollo sobre el sistema), o cubrir totalmente una asignatura común de sistemas operativos.

En el libro aparecen ejemplos ejecutados en el terminal, por ejemplo:

```
 $  echo esto es un ejemplo
esto es un ejemplo
 $
```

En todos los ejemplos de terminal, `$` es el símbolo de sistema o *prompt*. Las líneas que escribe el usuario en el terminal para que se ejecuten aparecen en color `verde azulado`. La salida de los programas ejecutados aparecen en color `negro` (como la cadena "esto es un ejemplo" en el ejemplo anterior). En el texto, los nombres de comandos y las líneas de comando también se marcarán en `verde azulado`.

Al final de cada capítulo hay ejercicios no resueltos. Estos ejercicios pueden ser de diferente tipo: comandos, pruebas, desarrollo de programas en C o shell, etc. No todos los ejercicios son iguales. Están marcados con un nivel de dificultad, † (†: fáciles, ††: intermedios, † † †: avanzados) y tiempo ∗ (∗: menos de 3 horas ∗∗: entre un día y una semana, ∗ ∗ ∗: proyecto). Estas estimaciones están hechas para un alumno que está aprendiendo C y shell, y siguiendo la asignatura. Por ejemplo, para un programador experto, los tiempos y la dificultad cambian radicalmente. Te recomendamos que intentes hacerlos *todos*.

1 Introducción y estructura del sistema

1.1. Historia

En la primera era de los computadores digitales (aproximadamente de 1945 a 1955), los ordenadores eran del tamaño de una sala completa. Al principio, esas computadoras se construían con relevadores electromecánicos (o relés), hasta que se empezaron a utilizar las válvulas de vacío (también llamados tubos de vacío). En esos tiempos, un programa se elaboraba desde cero, escribiendo su patrón de bits.

Más tarde (de 1955 a 1965) aparecieron los lenguajes de programación como FORTRAN y los sistemas operativos primigenios que cumplían básicamente la función de una biblioteca: se podía reutilizar código. En esa época aparecieron los sistemas integrados y los sistemas por lotes.

Imagen (c) chipsetc.com

Figura 1.1: Una válvula 408A, un transistor de germanio, un transistor de silicio y un circuito integrado.

Posteriormente (alrededor de 1965-1980) se desarrollaron los lenguajes de programación de alto nivel como COBOL y C. En esa época aparecen computadores más pequeños, los minicomputadores (*minicomputer*), la *multiprogramación* (capacidad para ejecutar múltiples programas simultáneamente) y los sistemas operativos evolucionaron. En 1965 apareció *MULTICS* y en 1966 los sistemas de la familia *360* de IBM.

Estos sistemas ya proporcionaban *tiempo compartido* (*time-sharing*). Permitían a varios usuarios acceder al sistema y utilizarlo de forma *concurrente*, esto es, *"a la vez"*. En 1969 se desarrolla un sistema llamado *unics* en Bell Labs (AT&T). Ese sistema operativo evolucionó y se llamó posteriormente *Unix*.

Unix fue creado por Ken Thompson y Dennis Ritchie, inicialmente para una *minicomputadora DEC PDP-7* como la de la Figura 1.2. El sistema era peculiar en varios

Imagen (cc) Wikipedia

Figura 1.2: Ordenador para el que se empezó a desarrollar *unics*.

sentidos. Buscaba la simplicidad, usaba texto plano como formato principal para los datos, el acceso a ficheros era mucho más sencillo que en sistemas contemporáneos y seguía una filosofía muy particular: tener programas pequeños que hagan bien su trabajo y que puedan conectarse entre ellos para realizar tareas complejas. El sistema se hizo *portable*[1] y empezó a tener éxito. En 1972 los autores declaraban: *"the number of Unix installations has grown to 10, with more expected..."*.

A partir de 1980 aparecen los ordenadores personales, las estaciones de trabajo, explota la implantación de las redes de ordenadores (redes de área local, etc.), se llega a un alto nivel de integración, aparecen distintos paradigmas de lenguajes de alto nivel y se desarrollan una gran cantidad de sistemas operativos modernos. Muchos de ellos, descendientes de ese sistema llamado *Unix*[2].

En esa época había dos familias diferenciadas de sistemas Unix: los sistemas de la Universidad de Berkeley, *BSD (**B**erkeley **S**oftware **D**istribution)* y derivados, y el sistema desarrollado por la empresa AT&T, *Unix System V* y derivados. Esos sistemas ofrecían distintas interfaces. A finales de la década de 1980, se empezó ha hacer un esfuerzo por estandarizar los sistemas de tipo Unix con un estándar llamado *POSIX* (que tiene distintas versiones).

[1] Un sistema portable es aquel que puede funcionar sobre distinto hardware.

[2] En la bibliografía se puede ver el nombre UNIX (en mayúsculas) o Unix. El primero se suele usar para identificar la marca registrada o para referirse al sistema original. El segundo se suele usar de forma genérica para referirse a un tipo de sistema (o también para referirse al sistema original). En el libro,

Imagen (cc) Wikipedia. Autor: Peter Hamer

Figura 1.3: Ken Thompson y Dennis Ritchie programando una PDP-11, 1970.

En esa época se empezaron a crear múltiples sistemas operativos derivados de Unix o reimplementaciones de las mismas ideas. A estos sistemas se les llama, en general, *Unix-like*. ¿Cuántos sistemas Unix o *Unix-like* hay? Muchos. En la Figura 1.4 puedes ver un esquema simplificado. Si quieres ver todos los sistemas, puedes encontrar una línea temporal completa en https://www.levenez.com/unix/.

Algunos de los sistemas operativos actuales más populares son sistemas de este tipo: GNU/Linux, Android, Mac OS X, iOS, OpenBSD o FreeBSD. Otros sistemas operativos, como los de la familia Microsoft Windows, no son derivados de Unix (aunque incorporan ideas, han tenido *subsistemas* compatibles con POSIX y desde Windows 10, los sistemas de Microsoft incorporan un Linux virtualizado llamado *WSL (**W**indows **S**ubsystem for **L**inux)*.

Si te interesa la historia de Unix, debes leer el libro de memorias de Brian Kernighan *Unix: A History and a Memoir* [4]. En este libro usaremos el sistema *Unix-like* más popular en la actualidad: GNU/Linux.

---

a partir de ahora, usaremos la segunda forma genérica.

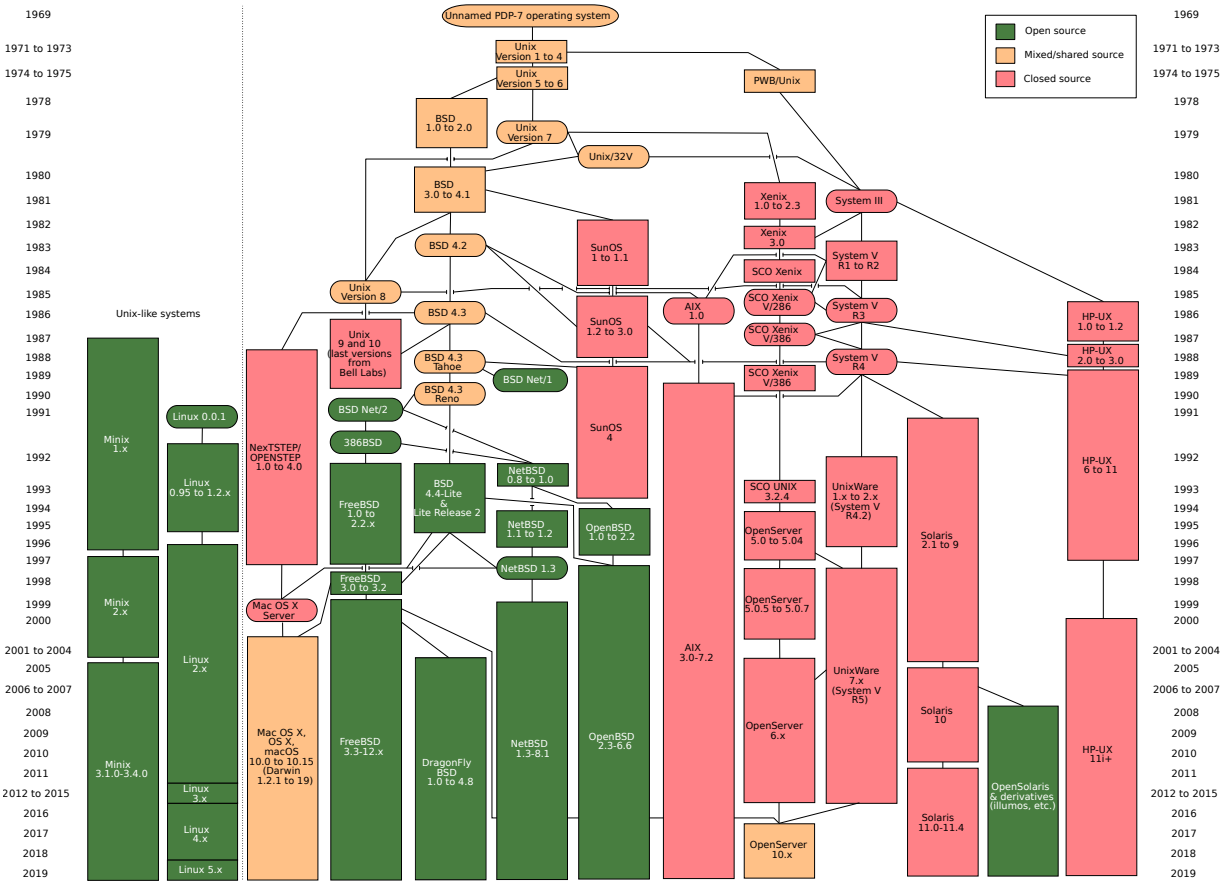

Imagen (cc) Wikipedia. Autores: Eraserhead1, Infinity0, Sav_vas

Figura 1.4: Historia simplificada de los sistemas de tipo Unix.

1.2. ¿Qué es un sistema operativo?

No existe una definición canónica para *sistema operativo*. En distintas referencias puedes encontrar diferentes definiciones. Es algo subjetivo y relativo. Informalmente, se puede definir como el conjunto de programas que permite usar la máquina.

El sistema operativo es como una biblioteca: implementa las partes comunes que todos los programas necesitan para usar el hardware. Pero va más allá:

- Gestiona y reparte la máquina. Hace lo necesario para que distintos programas puedan ejecutar sobre el mismo hardware. *Multiplexa* la máquina en tiempo (p. ej. la CPU) y en espacio (p. ej. la memoria).

- Abstrae la máquina. Ofrece una *máquina virtual*: una máquina ficticia, inventada por el software. Oculta los detalles de la máquina real y la complejidad que supone compartirla entre distintos programas en ejecución. De esta forma, podemos escribir nuestros programas sin estar pendientes de cómo se reparten los recursos (CPU,

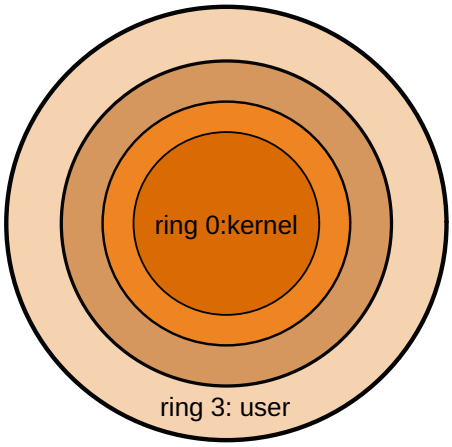

Figura 1.5: Tradicionalmente sólo se usan dos anillos de privilegio de la CPU: ring 0 para el kernel y ring 3 para los programas de usuario.

memoria, disco, etc.), como si nuestro programa ejecutase en una máquina propia en la que únicamente está él. En este libro vamos a estudiar distintas abstracciones proporcionadas por el sistema operativo: ficheros, directorios, conexiones, procesos, etc.

El *núcleo del sistema operativo* (*kernel*) es el programa que se encarga de hacer todo eso. No es magia. Es un programa, normalmente escrito en C y en ensamblador, capaz de ejecutar todas las instrucciones de la CPU. El resto de programas son *programas de usuario*: programas que ejecutan en *área de usuario* (*userland, userspace*). Esos programas sólo pueden ejecutar un subconjunto de las instrucciones de la CPU. Cuando necesitan hacer algo que no pueden, tienen que pedírselo al núcleo del sistema. Los programas de usuario están formados por su propio código y el código de las bibliotecas que usan (p. ej. para implementar la interfaz gráfica, operaciones fundamentales, seguridad, etc.). La Figura 1.6 muestra la estructura del sistema.

Las CPUs tienen niveles de privilegio o *anillos (rings)*. El tipo de instrucciones, los registros y las zonas de memoria a las que puede acceder un programa depende del anillo en el que se encuentre ejecutando.

Tradicionalmente, en los sistemas de tipo Unix, se usan dos anillos. Los procesadores Intel tradicionalmente tenían cuatro anillos, del 0 al 3:

- Ring 3: programas de usuario, no pueden usar las instrucciones para gestionar los recursos.
- Ring 0: núcleo del sistema, puede usar las instrucciones para gestionar los recursos.

Los anillos 1 y 2 no se suelen utilizar (algunos sistemas sí los han usado, pero no suele ser lo habitual). Los procesadores modernos tienen rings por debajo del 0: -1 (Intel VT-x, AMD-V) para ejecutar las instrucciones que dan soporte para la virtualización, -2 para

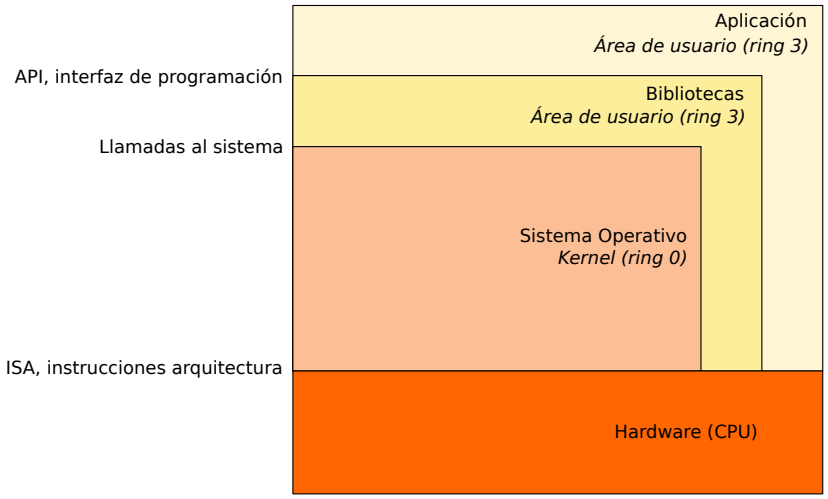

Figura 1.6: Estructura del sistema.

las instrucciones que ejecuta el *firmware* para la gestión de energía, etc. (*SMM o **S**ystem **M**anagement **M**ode*) y -3 para el llamado Intel ME (sin documentar, procesador propio que inicializa el procesador principal y monitoriza el sistema).

1.2.1. Distribuciones

Una distribución es una colección concreta de software de área de usuario y un núcleo del sistema operativo. Hay distribuciones de distintos sistemas. Linux tiene muchas distribuciones distintas, agrupadas en familias (unas 600 distribuciones).

Las que usan el kernel de Linux y las herramientas de área de usuario de GNU (comandos, biblioteca estándar, etc.) se denominan distribuciones GNU/Linux. Por ejemplo, las más populares son Debian, Ubuntu, Red Hat y SUSE. Hay distribuciones con soporte comercial, como Ubuntu o Red Hat. También hay distribuciones generalistas y otras dirigidas a nichos: seguridad (Kali), privacidad (Tails), comunicaciones (OpenWrt), ocio (Kodi), especializadas para distintas plataformas (Raspian para Raspberry Pi), etc. Los ejemplos de este libro los hemos probado sobre una distribución Ubuntu.

Una distribución suele tener su sistema de gestión de paquetes para instalar el software. Por ejemplo: `apt` es el sistema de paquetes de las distribuciones basadas en Debian, incluyendo Ubuntu (los paquetes son ficheros `.deb`). Otro ejemplo es `rpm`, que es el sistema de paquetes de las distribuciones basadas en Red Hat.

Las distribuciones difieren en distintos aspectos aunque usen el mismo núcleo y compartan muchas de las herramientas y aplicaciones. Por ejemplo, las distribuciones suelen tener distintos programas de instalación, una estructura del árbol de ficheros ligeramente diferente, algunas herramientas específicas, software propietario, etc. Hay que tener en cuenta que cada distribución tiene su propio esquema de versiones (números, versiones con soporte a largo/corto plazo, etc.). Hay un intento de estandarización del esquema

de ficheros, comandos, partes del sistema y otros detalles de administración que se llama *LSB (**L**inux **S**tandard **B**ase)*.

Cuando elegimos una distribución, necesitamos saber la arquitectura de la máquina (AMD64, x86, ARM, etc.). No todas las distribuciones están disponibles para todas las arquitecturas.

1.2.2. Núcleo

El núcleo o *kernel* del sistema operativo es un programa que ejecuta en **modo privilegiado** (ring 0). Esto significa que puede ejecutar instrucciones especiales (acceder a ciertos registros de la CPU, invalidar cachés, etc.).

Su tarea fundamental es multiplexar la máquina (espacio y tiempo): implementa las **políticas y mecanismos** para repartir la CPU, memoria, disco, red, etc.

También se encarga de manejar el hardware. Los **drivers** son partes del núcleo que se pueden instalar y contienen el código necesario para manejar un dispositivo. Por esa razón, cuando instalamos un nuevo hardware en nuestro ordenador (p. ej. una tarjeta de video nueva), necesitamos instalar su driver.

Como ya hemos dicho antes, el núcleo del sistema proporciona **abstracciones**, como los procesos (programas en ejecución), ficheros (datos agrupados bajo un nombre), etc.

El núcleo da servicio al resto de programas en ejecución, que no ejecutan en modo privilegiado. Si el kernel es **reentrante**, puede dar servicio a múltiples programas simultáneamente. Esto puede significar, por ejemplo, interrumpir al kernel ejecutando en el contexto de un proceso para ejecutar otro (pseudoparalelismo) o tener ejecutando en el kernel dos procesadores en el contexto de procesos simultáneamente (paralelismo), explicaremos esto en detalle más adelante. Todos los kernels de tipo Unix modernos son reentrantes.

Los módulos del kernel son subprogramas que actúan como *plugins* del núcleo: se pueden cargar mientras el núcleo está en ejecución. Esto se llama carga dinámica. La mayoría de los kernels actuales permiten la carga dinámica de *módulos* para ampliar o reducir su funcionalidad sin la necesidad de rearrancar el sistema. Esto permite cargar un *driver*, un protocolo de red o el soporte para un sistema de ficheros concreto sólo cuando es necesario usarlo, sin tener que reiniciar el sistema. Los ficheros que son módulos del kernel tienen un nombre diferente en cada sistema operativo: en Linux terminan (o tienen *extensión*[3]) en `.ko`, en Mac OSX en `.kext`, en FreeBSD en `.kld`, en Windows su extensión es `.sys`, etc.

En Linux, los módulos se gestionan con los siguientes comandos:

- `lsmod` escribe en su salida la lista de módulos cargados actualmente.

[3]Siendo precisos, la extensión es un metadato que sólo existe en algunos sistemas operativos (p. ej. Windows.). En los sitemas de tipo Unix, los ficheros no tienen un metadato llamado extensión. Sin embargo, se suele decir que la *extensión* es la parte del nombre que va después del último punto. Esa parte del nombre se usa, como convenio, para dar pistas sobre el contenido del fichero. Por ejemplo, lo suelen usar los exploradores de ficheros para dibujar el icono que representa al fichero (p. ej. un icono de *PDF* cuando el fichero termina en `.pdf`). De aquí en adelante, cuando hablemos de la *extensión* de un fichero, nos estamos refiriendo a la parte de su nombre después del último punto.

- `modprobe` carga un módulo.
- `rmmod` descarga un módulo.
- `modinfo` ofrece información sobre un módulo (autores, licencia, fichero, etc.).

Los ficheros de los módulos están en[4]:

`/lib/modules/`*`versión-de-kernel`*`/`

Por ejemplo:

```
 $  uname -r
5.6.0-1028-oem
 $  modinfo --filename  intel_lpss
/lib/modules/5.6.0-1028-oem/kernel/drivers/mfd/intel-lpss.ko
 $
```

Tipos de núcleos

Respecto a su estructura interna, un núcleo puede ser *monolítico* o *microkernel*.

Los *monolíticos* se implementan como un único programa. Son sencillos y rápidos en general. El problema es que no hay protección entre distintos componentes del núcleo y puede haber falta de estructura (depende de la implementación del sistema). Dos ejemplos de núcleos monolíticos son Linux y FreeBSD.

Los *microkernels* se basan en tener un núcleo muy pequeño, reducido a lo mínimo: abstracción del hardware, flujos de control, comunicación y gestión de memoria. El resto (lo que se llama *OS personality*: gestión de procesos, red, sistemas de ficheros, etc.) se implementa en componentes[5] independientes. La idea fundamental es separar las **políticas** en espacio de usuario y dejar los **mecanismos** en espacio de kernel[6].

Las ventajas de un microkernel son la modularidad y la tolerancia a fallos en los componentes. El problema es que es complicado que sean tan eficientes como los núcleos monolíticos por la comunicación entre sus componentes. Los primeros microkernels eran

[4] Podemos ver la versión del kernel que estamos ejecutando ahora mismo ejecutando el comando `uname -r`

[5] Esos componentes de sistemas con microkernel se denominan *servidores*, pero no se deben confundir con los programas de usuario también llamados servidores o servicios, como por ejemplo un servidor web (p. ej. `httpd`).

[6] Esta idea es importante en general a la hora de programar. Hay que separar las políticas (decisiones sobre cómo usar y gestionar los recursos) de los mecanismos (abstracciones que permiten a las políticas tomar las acciones necesarias para ejecutar dichas decisiones). Separar los mecanismos permite cambiar las políticas (o tener varias) sin tener que cambiar todo el código, aumentando la modularidad. Esta separación aparecerá mucho a la hora de hablar de sistemas operativos, ya que el sistema operativo implementa mecanismos para delegar en el usuario las decisiones y políticas para repartir y gestionar el uso de recursos entre diferentes usuarios y programas.

demasiado lentos. Las nuevas generaciones de microkernels tienen mejor rendimiento. Por ejemplo, Mach, L4 y derivados son microkernels.

En ocasiones, se sigue un esquema mezcla de las dos aproximaciones. Un kernel *híbrido* es un compromiso entre microkernel y kernel monolítico. Algunos incluyen ciertos componentes en espacio de kernel (dejando de ser tan *micro*), como los drivers, gestión de procesos, etc. Dos ejemplos de estos sistemas son Minix y QNX.

Otros núcleos llamados *híbridos* simplemente siguen un diseño de microkernel, pero en realidad todos los servidores se enlazan con el núcleo y ejecutan en espacio de kernel, como en un núcleo monolítico. Son microkernel *en espíritu*. Dos ejemplos son XNU (el núcleo de Mac OS X) y Windows NT (el núcleo de Windows 10).

1.2.3. Área de usuario

Los programas que utilizamos normalmente (p. ej. las aplicaciones) ejecutan en área de usuario. Otros programas de usuario son herramientas del propio sistema operativo, como la *shell* (intérprete de comandos), los comandos de administración o la interfaz gráfica de usuario. Aunque algunas personas consideran que el *sistema operativo* es sólo el *núcleo*, en realidad un sistema operativo está formado por más componentes.

Todos estos programas se ejecutan en **modo no privilegiado** (ring 3), no pueden ejecutar instrucciones peligrosas (esto es, las que permiten gestionar los recursos de la máquina).

Cuando necesitan ejecutar alguna acción que requiere usar instrucciones privilegiadas, los programas de usuario piden servicio al kernel realizando **llamadas al sistema**. Como se observa en la Figura 1.6, las aplicaciones pueden usar el API de las bibliotecas y las llamadas al sistema para pedir servicio al núcleo. De cara al usuario, una llamada al sistema es como una llamada a una función de una biblioteca. En realidad, es algo más complicado.

1.2.4. Flujo de ejecución

La abstracción del sistema operativo para un flujo de ejecución (o control) se llama proceso. En general, estos son los dos elementos fundamentales para un flujo de ejecución:

- **Contador de programa**: un registro de la CPU[7](*PC*, ***P**rogram **C**ounter*) que apunta a la instrucción por la que va ejecutando el programa.

- **Pila**: un registro de la CPU (*SP*, ***S**tack **P**ointer*) que apunta a la zona de la memoria donde se guardan los **registros de activación** (o marcos de pila). Estas estructuras guardan los datos necesarios para realizar llamadas a procedimiento (parámetros, variables locales, dirección del programa para retornar, etc.).

[7]El registro puede tener distinto nombre dependiendo de la arquitectura, por ejemplo en Intel de 64 bits este registro se llama `RIP` y apunta al final de la instrucción actual (o inicio de la siguiente). En otras arquitecturas es diferente.

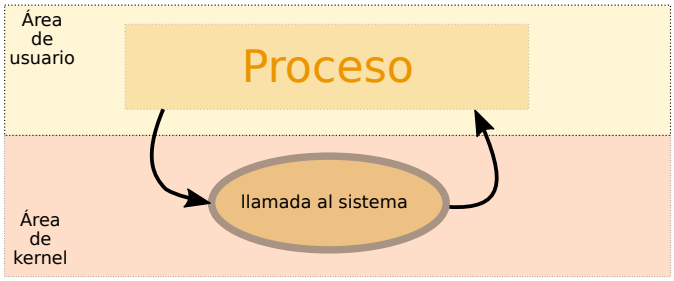

Figura 1.7: Llamada al sistema

Estos son los dos datos básicos que forman parte del **contexto de un proceso**. Son la abstracción básica de la CPU. Aunque hay otros registros (que contendrán datos con cálculos intermedios), si elegimos bien el punto del programa (y no hay cálculos a medio hacer), estos registros abstraen totalmente el estado de la CPU asociada al proceso. En realidad, un proceso tiene bastantes más elementos asociados en la estructura de datos que lo representa dentro del núcleo, pero con estos dos ya podemos tener un flujo básico de ejecución que se puede conmutar.

Se dice que los flujos de ejecución son **concurrentes** cuando varios están ejecutando *"a la vez"*. Esto puede pasar cuando tenemos más de una CPU (paralelismo) o cuando el sistema reparte en el tiempo una única CPU entre distintos flujos (pseudoparalelismo). En cualquier caso, el sistema operativo crea la ilusión de que cada proceso tiene su propia CPU. Se dice que una interfaz es **concurrente** cuando el programador no puede distinguir entre esos dos casos (esto es, las operaciones que puede usar son las mismas independientemente del tipo de paralelismo), aunque sí puede notar una diferencia de rendimiento en la práctica.

1.2.5. Llamadas al sistema

Como ya se ha comentado, cuando un programa de usuario necesita que se realicen acciones que requieren instrucciones privilegiadas, realiza una llamada al sistema.

En este caso, *entra al kernel* por iniciativa propia: el proceso deja de ejecutar el código de la aplicación y pasa a ejecutar, en modo privilegiado, el código del kernel. Cuando pasa esto, se dice que se ejecuta el código del kernel *en el contexto del proceso*. Por eso, el proceso tiene en realidad dos pilas: su pila de usuario (para implementar llamadas a procedimiento cuando ejecuta normalmente en área de usuario) y su pila de kernel (para implementar llamadas a procedimiento cuando ejecuta el núcleo en el contexto de un proceso).

Veamos cómo funciona una llamada al sistema. Este ejemplo (simplificado) corresponde al sistema operativo Plan 9[8] en la arquitectura AMD64:

[8] Plan 9 es un sistema operativo de investigación que desarrollaron en Bell Labs los desarrolladores del Unix original y en el que han colaborado los autores de este libro. Está programado de forma muy limpia y lo usaremos como ejemplo de implementación cuando sea relevante. Aunque es parecido a Unix, tiene importantes diferencias. Otro sistema operativo de código limpio y fácil de leer y más

1. Coloca los argumentos en la pila del proceso.
2. Carga el número de llamada al sistema en un registro.
3. Ejecuta la instrucción `SYSCALL`, que pasa a ring 0 y salta al punto de entrada de las llamadas al sistema en el kernel (apuntado por el registro `Lstar`, que habrá sido configurado en tiempo de arranque del sistema operativo). A continuación el kernel hace lo siguiente:
   a) Cambia el Puntero de Pila (SP) para usar pila de kernel del proceso.
   b) Guarda el contexto del proceso, su estado en área de usuario, en la pila de kernel. El contexto contiene, entre otras cosas, los registros de la CPU, incluyendo el puntero de pila de área de usuario y el contador de programa por el que va ahora mismo ejecutando las instrucciones del programa de usuario.
4. Copia los argumentos de la llamada al sistema a la estructura que representa al proceso.
5. Se llama a la función que implementa la llamada al sistema dentro del kernel. Para ello, se usa el número dejado en el registro (en el paso 2) para indexar un array que tiene todas las llamadas al sistema (tabla de llamadas al sistema).
6. Ejecuta la llamada al sistema.
7. Restaura el contexto del proceso y ejecuta la instrucción `SYSRET` para volver a ring 3.

1.2.6. Arranque del sistema

¿Cómo llegamos a tener el sistema completo ejecutando? Para ello hay que cargar el programa que ejecuta en modo privilegiado (el *kernel*). Esto puede ser un proceso complicado porque antes hay que inicializar el hardware (pasar el procesador a 64 bits de un modo de arranque de compatibilidad de 16 o de 32 bit, inicializar el subsistema de memoria, de almacenamiento secundario o de red para poder poner el programa a ejecutar, etc.).

Grosso modo, el proceso de arranque de un sistema es este:

1. Al arrancar la máquina, se salta a una posición de memoria concreta. En esa posición suele haber proyectado en memoria un programa suministrado por el fabricante de la placa madre, que viene almacenado en una ROM (o memoria *Flash*): el *firmware*. Básicamente, ese programa inicializa el hardware y carga el *cargador primario*. Un ejemplo de ésto es *EFI* (***E**xtensible **F**irmware **I**nterface*) en un PC[9] o *U-boot* en muchos dispositivos con procesador ARM.

cercano a Linux (es un Unix) es OpenBSD.

[9] Antes este firmware se llamaba BIOS. En los PCs modernos, el firmware es UEFI, y la forma en la que está programada es diferente.

2. El cargador primario seguramente cargue otros cargadores secundarios. El objetivo final de la cadena de cargadores es inicializar el hardware lo suficiente como para encontrar el fichero binario del kernel y cargarlo en memoria principal (RAM). Normalmente el fichero del kernel vendrá del disco duro, pero también puede venir por la red o de un disco externo. Ejemplos de cargadores (incluyen primario y secundario) son *grub* y *LILO* (***Li**nux **Lo**ader*). El cargador de Windows se llama *Windows Loader* (el proceso de carga de ese sistema es bastante complicado, usa varios cargadores encadenados).

3. Una vez cargado el núcleo, se empieza a ejecutar desde su punto de entrada. Se inicializa el núcleo (varias fases distintas), se configura el sistema de ficheros principal, etc. Al final se crean los primeros procesos de usuario, entre ellos un proceso bastante importante llamado *init*.

4. Ese programa *init* va creando los procesos para ejecutar los programas de usuario (shells, interfaz gráfica, servidores, etc.)[10]. Uno de esos programas nos dejará introducir el nombre de usuario y autenticarnos para comenzar a trabajar.

A partir de ese momento, se irán creando todos los procesos de usuario necesarios para los programas que ejecutemos.

1.3. Virtualización

Hoy en día es común usar distintos esquemas de virtualización. En general, existen distintos tipos de *máquinas virtuales* [5]:

- **Máquina virtual de proceso**: tiene como objetivo proporcionar una plataforma para ejecutar un único programa. Por ejemplo, pueden ser emuladores de otra ISA[11] real (p. ej. OSX Rosetta), optimizadores o emuladores de ISAs virtuales (p. ej. Java VM, .NET).

- **Máquina virtual de sistema**: Se llama hipervisor (*hypervisor* o VMM) al programa que gestiona las máquinas virtuales para que ejecuten uno o varios sistemas sobre el mismo hardware. Proporciona un entorno completo y persistente para ejecutar un sistema operativo completo. El trabajo de un hipervisor es muy parecido al de un kernel: multiplexar la máquina (pero para usar distintos sistemas encima).

A continuación veremos cómo se sitúa el sistema operativo en los distintos esquemas de virtualización. Durante el resto del libro, nos centraremos en una estructura simple como la presentada en la Figura 1.6.

[10] En los sistemas GNU/Linux actuales, `systemd` es el proceso que hace el papel de init.
[11] La ISA, **I**nstruction **S**et **A**rchitecture es el conjunto de instrucciones de un procesador.

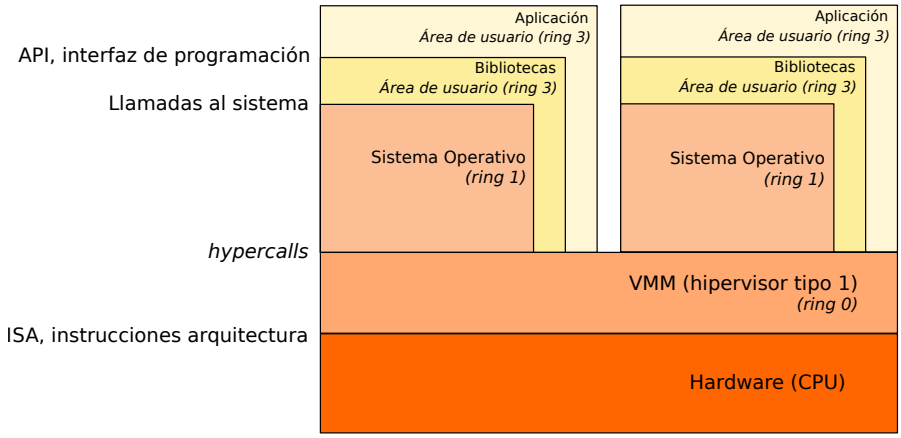

Figura 1.8: Estructura con paravirtualización.

Máquina virtual clásica (hypervisor tipo 1)

Este tipo de máquinas virtuales también se llaman *bare metal* o *unhosted*. Hay dos subtipos:

- **Paravirtualización**. El sistema operativo huésped tiene que estar modificado para ejecutar sobre un hipervisor de este tipo. El huésped realiza hiperllamadas (*hypercalls*) para pedir servicio al hipervisor, por ejemplo para gestionar la tabla de páginas, configurar el hardware, etc.

  El concepto de *hypercall* es similar al de llamada al sistema (pero a distinto nivel). Por ejemplo, Xen y KVM proporcionan este tipo de virtualización.

- **Virtualización completa asistida por hardware**. Se basa en instrucciones especiales de la CPU para virtualización, como Intel VT-x o AMD-V.

  Esas instrucciones especiales sirven para activar el modo VMX root (Ring -1), lanzar una máquina virtual, pasar el control al hipervisor, retomar una máquina virtual, etc. En este caso, el sistema operativo huésped no necesita modificaciones. Un ejemplo de este tipo de virtualización es vSphere.

Máquina virtual alojada (hypervisor tipo 2)

En este caso, la máquina virtual se aloja sobre otro sistema operativo. Por ejemplo, podemos crear una máquina virtual Windows encima de un sistema Linux.

El hipervisor puede instalar drivers en el sistema operativo anfitrión para mejorar el rendimiento. Algunos ejemplos son VMWare Fusion, Virtual Box o QEMU.

Si se habla de una *Whole-system VM* también se emula la ISA de otro procesador. Virtual PC es un ejemplo (QEMU también puede hacer esto).

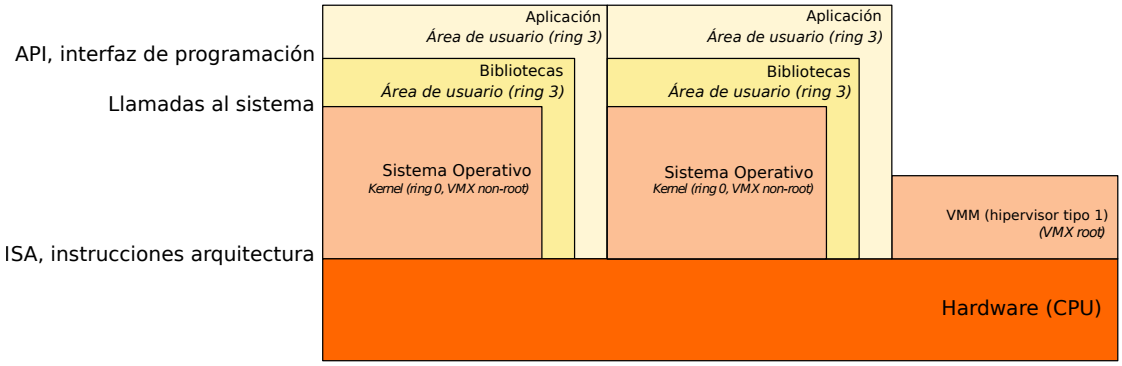

Figura 1.9: Estructura con virtualización asistida por hardware.

Contenedores

Una máquina virtual aisla distintas **imágenes completas** de distintos sistemas operativos ejecutando. Si lo que queremos aislar es un servicio, pagamos cierto coste ejecutando un sistema operativo completo para él (tiempo en arrancar y parar la máquina virtual, rendimiento, etc.).

Hay otra forma de virtualizar: los **contenedores**. Esto se denomina *virtualización a nivel del sistema operativo*. En este caso, en el propio kernel se pueden crear distintos entornos aislados, cada uno con sus propias abstracciones y recursos (espacio de procesos, sistema de ficheros raíz, CPU, recursos de red, usuarios, etc.).

Los contenedores tienen varias ventajas sobre las máquinas virtuales. El arranque de un contenedor es muy rápido (consiste en configurar únicamente el espacio de usuario). También son más ligeros, ya no se necesita una imagen entera del sistema.

Las desventajas son que hay menos aislamiento y, por tanto, menos seguridad. Además, como es evidente, no es posible virtualizar distintos sistemas operativos (p. ej. Linux y Windows), porque los contenedores comparten el núcleo. Por ejemplo, Docker, OpenVZ y Linux Containers (LXC) son herramientas para gestionar contenedores en Linux.

Ahora ya podemos situar al sistema operativo en el mapa para todos los casos. Como se dijo anteriormente, a partir de ahora nos centraremos en una estructura simple como la presentada en la Figura 1.6.

1.4. Introducción al uso de un sistema de tipo Unix

Para usar un sistema de tipo Unix eficientemente hay que usar el intérprete de comandos. Un *comando* o mandato (*command*) es una cadena de texto que identifica a un programa u orden.

El intérprete de comandos o *shell* es un programa cuya función es ejecutar comandos de forma interactiva. Este tipo de herramientas también se suelen llamar *CLI* (***C**ommand **L**ine **I**nterface*). Por lo general, también permiten crear programas (*scripts*) en un lenguaje propio. Hay distintos tipos de shells. En Linux, lo habitual es usar `bash`.

Básicamente, una shell hace esto en un bucle infinito:

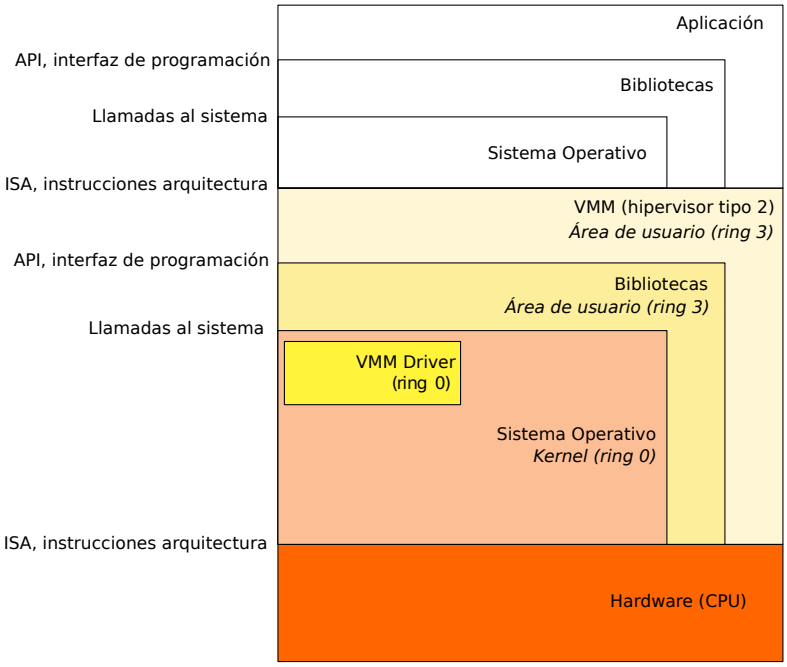

Figura 1.10: Estructura con máquina virtual alojada.

1. Mostrar el *prompt*, el texto que indica que la shell está esperando una orden.

2. Leer un línea de comandos.

3. Sustituir algunas cosas en esa línea.

4. Crear los procesos para ejecutar los comandos descritos por la línea.

De momento, si quieres probar los comandos, ejecuta una aplicación llamada `terminal` y dentro podrás ejecutar comandos de shell. El Capítulo 2 está dedicado entero al uso de la shell y allí se explica más en detalle todo esto.

Cuando entramos al sistema, lo hacemos con un *usuario* (*login name*). Todos nuestros programas ejecutarán a nombre del usuario con el que entramos al sistema. Para entrar al sistema, hay que introducir el nombre de usuario y la contraseña asociada. El sistema autenticará al usuario y abrirá una sesión. Entonces ya se puede usar un terminal para empezar a trabajar con el sistema.

En Unix hay un usuario especial llamado `root`. Es el superusuario o administrador, y tiene permisos para hacer lo que quiera en el sistema. No es buena idea trabajar como root. Esa cuenta sólo se debe usar cuando es necesario tener permisos especiales para administrar el sistema. Durante el resto del tiempo debemos usar una cuenta normal.

Los usuarios normales suelen tener permiso para trabajar únicamente con sus datos, que estarán en su directorio casa (*home*).

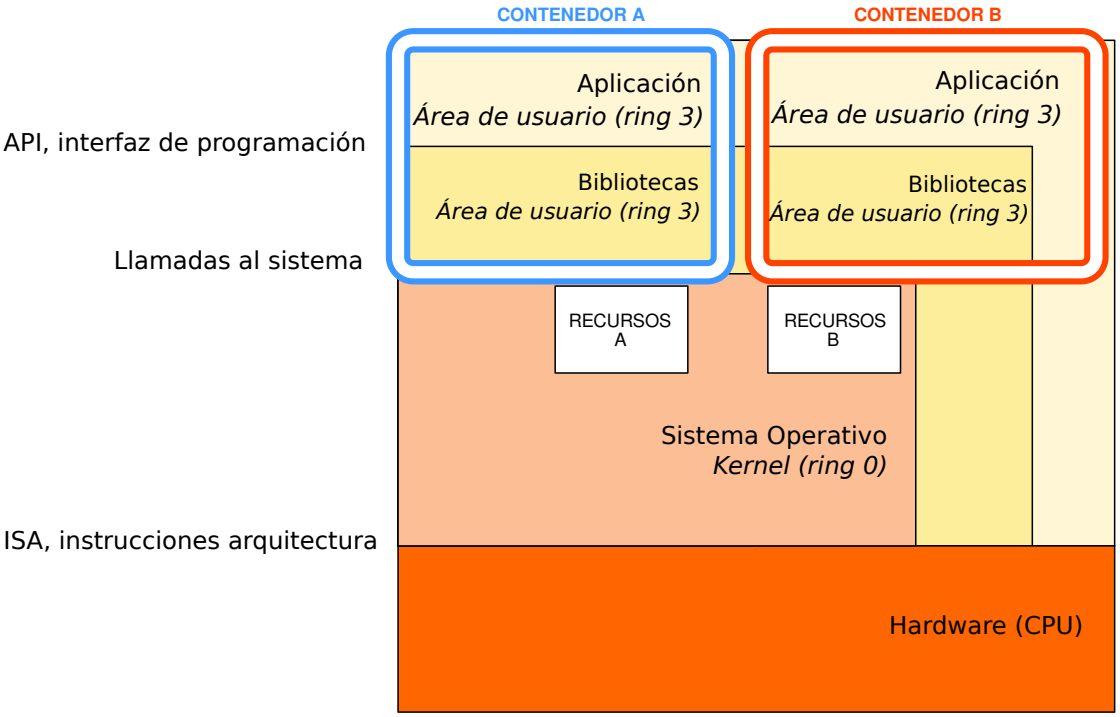

Figura 1.11: Estructura con contenedores.

1.4.1. Árbol de ficheros

Los datos están organizados en árboles de ficheros, que como ya hemos visto, es una abstracción que proporciona el sistema.

En Unix, un *fichero* no es más que una secuencia de bytes con un nombre dado (y otros metadatos asociados, como tamaño, usuario al que pertenece, etc.). Se llama *directorio* a los objetos que contienen ficheros y otros directorios. Dos ficheros que están en distintos directorios son dos ficheros diferentes, independientemente de su nombre o contenido[12].

En otros sistemas, se habla de *archivos* y *carpetas* en lugar de ficheros y directorios. Son similares. Nosotros usaremos los nombres tradicionales en sistemas de tipo Unix.

Hay un directorio especial: el raíz (/). Es el directorio del que *cuelga* todo el árbol de ficheros. La Figura 1.12 muestra un ejemplo del árbol de ficheros.

El árbol de ficheros en un sistema GNU/Linux tiene estos directorios:

- `/bin` tiene ficheros ejecutables.
- `/dev` tiene dispositivos.
- `/etc` tiene ficheros de configuración.
- `/home` tiene los datos personales de los usuarios.

[12] A veces dos nombres se refieren a los mismos datos, como veremos en el capítulo 4

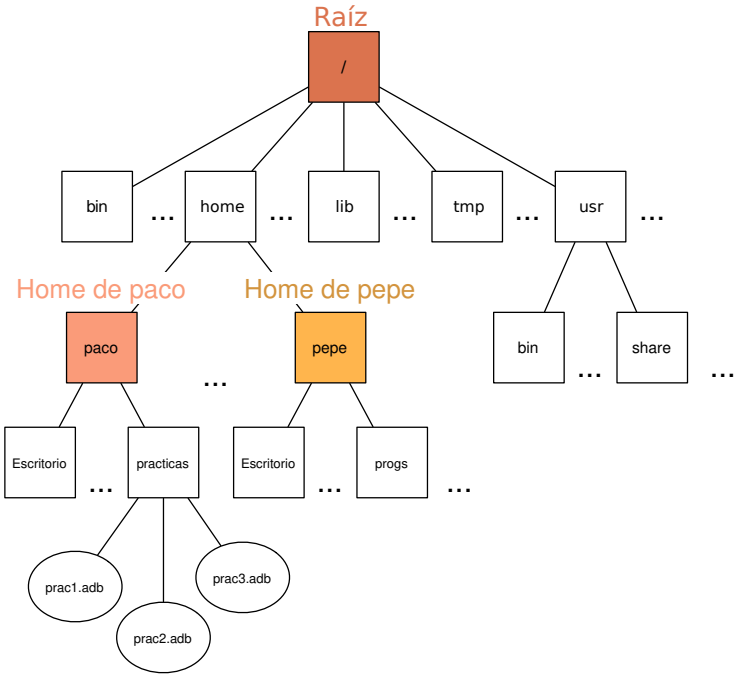

Figura 1.12: Árbol de ficheros.

- `/lib` tiene las bibliotecas (código) que usan los programas ejecutables.
- `/proc` y `/sys` ofrecen una interfaz de *ficheros sintéticos* (ya hablaremos de esto más adelante) para interaccionar con el núcleo del sistema.
- `/sbin` tiene los ejecutables del sistema.
- `/tmp` sirve para almacenar los ficheros temporales, se borra en cada reinicio.
- `/usr` existe por razones históricas (tamaño de almacenamiento) y contiene gran parte del sistema: contiene directorios similares a los anteriores (/usr/bin o /usr/lib), con los datos y recursos para los programas de usuario (no del sistema).
- `/var` tiene los datos que se generan en tiempo de ejecución (caché, logs y otros ficheros que generan los programas).
- `/boot` contiene los ficheros de arranque del sistema (cargadores, núcleo, etc.).
- `/media` y `/mnt` son puntos de montaje (donde aparecerán los sistemas de ficheros que provienen de otras particiones o discos).
- `/opt` contiene ficheros para programas *de terceros*.

Un programa en ejecución siempre tiene un directorio de trabajo. Es el directorio en el que está trabajando actualmente. En la shell, podemos saber el directorio de trabajo actual con el comando `pwd`[13]:

```
 $ pwd
/home/esoriano
 $
```

El directorio de trabajo en una shell recién creada es siempre el directorio casa del usuario.

Se llama *ruta* o *path* al camino para llegar a un fichero. Una **ruta absoluta** es la serie de directorios desde el raíz separados por barras para llegar al fichero. Una ruta absoluta siempre empieza con el raíz (/). Por ejemplo:

```
/home/pepe/fichero.txt
```

Una **ruta relativa** describe el camino con una serie de directorios desde el directorio actual. Por ejemplo, esta es la ruta para alcanzar el fichero anterior si nuestro directorio de trabajo actual es `/home`:

```
pepe/fichero.txt
```

Podemos usar el nombre de directorio . (punto) para referirnos al directorio de trabajo actual. También podemos usar .. (dos puntos) para hacer referencia al directorio padre del directorio de trabajo actual (esto es, el directorio que tiene dentro el directorio de trabajo actual).

Por ejemplo, ahora vamos a ver el contenido (usando el comando `cat`) del mismo fichero en todos los casos, pero usando distintas rutas:

```
 $ pwd
/home/esoriano
 $ cat hola.txt
hola
 $ cat /home/esoriano/hola.txt
hola
 $ cat /home/esoriano/../esoriano/hola.txt
hola
 $ cat /tmp/../home/./esoriano/hola.txt
hola
 $ cat ./hola.txt
hola
 $
```

[13] El prompt en los ejemplos del libro será `$ `.

Obsérvese que `.` y `..` se refieren al directorio actual (y padre respectivamente) de la ruta resuelta hasta el momento. Así, `..` se refiere al directorio padre del directorio actual de la shell (path relativo) pero `/home/pepe/..` se refiere a `/home`, es decir, el directorio padre de `/home/pepe` (que es la ruta resuelta hasta ese punto).

1.4.2. Manual

Hay un comando Unix que hay que aprender a usar antes que el resto de comandos: `man`. Ese comando sirve para consultar las páginas de manual, que es donde se explica cómo se usan los otros comandos, funciones y otras herramientas y ficheros de configuración.

Al comando `man` se le puede especificar la sección del manual y el asunto. Hay que tener en cuenta que distintas secciones del manual pueden tener páginas sobre un mismo asunto. Las secciones del manual se describen en la propia página de manual de `man`. La primera sección es de comandos, la segunda está dedicada a las llamadas al sistema, la tercera a las bibliotecas, etc. Para hacer referencia a una página de manual, se suele usar la notación *asunto(sección)*. Por ejemplo, *cat(1)*. A partir de ahora usaremos esta notación en el libro.

Por ejemplo, para ver el manual del programa `man` de la sección 1, esto es, *man(1)*:

```
$ man 1 man
```

El manual ejecuta un paginador que permite navegar por el texto. Acepta las siguientes teclas:

- **q** sale.
- / busca texto (hay que escribir eso y una palabra a buscar **/palabra**) y **n**, **p** repite la búsqueda hacia atrás o hacia delante.
- **AvPág** y **RePág** avanzan página a página.
- ↓ y ↑ avanzan línea a línea.

Otro ejemplo: para ver el manual de la función de biblioteca *printf(3)*, podemos escribir:

```
$ man 3 printf
```

Para tener una idea rápida de qué hace un comando, se puede utilizar el comando `whatis`:

```
$ whatis ls
ls (1)               - list directory contents
$
```

Para buscar sobre una palabra podemos usar el comando `apropos`. Este programa imprime una lista de páginas de manual que contienen la palabra sobre la que consultamos. Es muy útil cuando no nos acordamos del nombre de un comando, pero sí de lo que hace. Por ejemplo, si queremos buscar un comando que usa cifrado:

```
$ apropos encryption
cbc_crypt (3)         - fast DES encryption
cmsutil (1)           - Performs basic cryptograpic operations
des_crypt (3)         - fast DES encryption
DES_FAILED (3)        - fast DES encryption
des_setparity (3)     - fast DES encryption
e4crypt (8)           - ext4 filesystem encryption utility
ecb_crypt (3)         - fast DES encryption
gpg (1)               - OpenPGP encryption and signing tool
gpgsm (1)             - CMS encryption and signing tool
passwd2des (3)        - RFS password encryption
SM2 (7ssl)            - Chinese SM2 signature and encryption
symcryptrun (1)       - Call a simple symmetric encryption tool
xcrypt (3)            - RFS password encryption
xdecrypt (3)          - RFS password encryption
xencrypt (3)          - RFS password encryption
$
```

Es recomendable usar las páginas de manual en inglés, ya que en muchas ocasiones las páginas traducidas están desactualizadas o incompletas. En el Capítulo 2 se explicará cómo hacerlo.

1.4.3. Texto plano

Una de las características del Unix original era que los datos se trataban principalmente como texto plano. Hay distintos formatos para el texto plano, los más usados son:

- ASCII: usa 1 byte para representar 128 caracteres (7 bits). Representa los caracteres que se usan en el idioma inglés. El primer bit se usaba para corregir errores (al principio los ordenadores y comunicaciones eran menos fiables). Ahora se pone a 0 y se usan sólo los bits restantes.

  Podemos ver la tabla ASCII, los caracteres y su representación numérica en un terminal el comando `ascii` (ver *ascii(1)*).

- ISO-Latin 1 (8859-1): usa 1 byte para almacenar caracteres (8 bits). Podemos representar los caracteres ASCII y además otros caracteres (por ejemplo, vocales acentuadas, eñe, etc.). Para indicar el conjunto extendido se utilizar el primer bit a 1. Si el primer bit está a 0, el carácter se interpreta igual que en ASCII. Hay otros conjuntos para representar otros caracteres de otros idiomas (cirílico, etc.), pero se solapan. Hay que saber a que conjunto se refiere el texto para poder leerlo.

- UTF-8: puede usar 1, 2 o más bytes[14]. Es compatible hacia atrás con ASCII y permite representar los caracteres de todos los alfabetos del mundo. Podemos ver los caracteres y su representación numérica en un terminal mediante el comando `unicode` y detalles de la implementación en el sistema en la página de manual *unicode(7)*. Éste sistema es el que debemos utilizar siempre que sea posible.

Siempre, antes de comenzar a usar un editor de texto plano, hay que configurar correctamente el formato de texto.

En la tabla ASCII hay caracteres imprimibles y otros caracteres de control, por ejemplo:

- El carácter `'\n'` indica nueva línea en el texto[15]. En Unix, es un convenio usado en todos los programas, bibliotecas, etc. En otros sistemas operativos no tiene por qué ser así (Windows usa la secuencia `'\n\r'`).
- El carácter `'\t'` indica un tabulador.
- No hay carácter EOF (end of file): eso es una invención de algunos lenguajes de programación.

Para editar ficheros de texto en un terminal, tenemos que usar un editor en modo texto. Ten en cuenta que no siempre tendrás una interfaz gráfica. En muchas ocasiones necesitaremos modificar ficheros de texto en un terminal.

El editor por excelencia en una shell es `vi`. El editor `vim` es un editor basado en `vi` (implementa un superconjunto). En ciertos sistemas, `vi` es en realidad `vim`.

Este editor tiene fama de ser complicado. Es fácil entrar en `vi`, pero es difícil salir (existen multitud de chistes y *memes* que hacen referencia a esto). Tiene dos modos: modo inserción (para escribir) y modo comando. Para pasar a cada modo:

- **i** pasa a modo inserción si estamos en modo comando. Cuando pulsemos teclas, escribiremos texto en el editor.
- **ESC** pasa a modo comando si estamos en modo inserción. Cuando pulsemos teclas, se ejecutarán comandos en el editor, por ejemplo para guardar el fichero, salir del editor, etc.

Si estamos en modo comando, podemos ejecutar comandos como estos (ten cuidado, algunos comandos van precedidos por dos puntos):

[14] En realidad define dos respresentaciones, una externa y una interna. La externa, que es la de ancho variable sirve para guardar en ficheros, y ocupa menos. La interna, de ancho fijo es la que se usa muchas veces en los programas, porque resulta más fácil programar con ella. Para pasar de una a otra hay que decodificar las runas. Esto es una simplificación ya que hay elementos de control, composiciones, varias formas de describir la misma escritura, etc. Los sistemas de escritura humanos son complicados. Aun así, es mejor tener soporte parcial de UTF que nada.

[15] Esta notación viene de cómo se representan estos literales de caracter en C. Por supuesto, '\n' representa **un sólo caracter**, el caracter representado por el byte 10, según se puede consultar en la tabla ASCII.

- **:q!** sale del editor sin guardar.
- **:x** salva el fichero y sale (también se puede con **:wq**).
- **:w** salva el fichero.
- **:número** se mueve a esa línea del fichero.
- **i** inserta antes del cursor.
- **a** inserta después del cursor.
- **o** inserta en una línea nueva.
- **dd** corta una línea.
- **p** pega la línea cortada anteriormente.
- **h, j, k, l** mueve el cursor a izquierda, abajo, arriba, derecha.

Existen muchos otros comandos para buscar, sustituir texto, ejecutar comandos externos, etc. Los comandos anteriores son los básicos para sobrevivir usando `vi`.

Hay otros editores en modo texto, como por ejemplo `nano`, pero es recomendable familiarizarse con `vi`, ya que está presente en prácticamente todos los sistemas de tipo Unix. Una vez que aprendamos a usar `vi`, podremos usar cualquier Unix en modo texto, algo muy común cuando tenemos que trabajar conectados por la red o administrar servidores.

1.4.4. Comandos básicos

Hay muchos comandos en Unix. Al comenzar es importante empezar a familiarizarse con estos comandos:

- `cd`: cambia el directorio actual de la shell.
- `echo`: escribe sus argumentos por su salida.
- `touch`: cambia la fecha de modificación de un fichero. Si no existe el fichero, se crea.
- `ls`: lista el contenido de un directorio.
- `cp`: copia ficheros.
- `mv`: mueve ficheros.
- `rm`: borra ficheros.
- `mkdir`: crea directorios.

- `rmdir`: borra directorios vacíos.
- `date`: muestra la fecha.
- `who`: muestra los usuarios que están en el sistema.
- `whoami`: muestra tu nombre de usuario.
- `sort`: ordena las líneas de un fichero.
- `wc`: cuenta caracteres, palabras y líneas de ficheros.
- `fgrep, grep`: buscan cadenas dentro de ficheros (los veremos más en profundidad más adelante).
- `cmp, diff`: comparan ficheros.
- `cat`: escribe en su salida el contenido de uno o varios ficheros.
- `less`: permite leer un fichero de texto en el terminal usando *scroll*.
- `file`: da pistas sobre el contenido de un fichero.
- `od`: escribe en su salida el los datos de un fichero en distintos formatos.
- `head, tail`: escriben las primeras o últimas líneas del fichero en su salida.
- `tar`: crea un fichero con múltiples ficheros dentro (comprimidos o no).
- `gzip, gunzip`: comprime y descomprime un fichero.
- `top`: muestra los procesos y el estado de sistema.
- `reset`: restablece el estado del terminal.
- `exit`: la shell termina su ejecución.

En sus páginas de manual se describen sus opciones y formas de uso. Es necesario que practiques en una shell y te familiarices con estos comandos antes de continuar con el resto de capítulos.

1.4.5. Variables

En la shell podemos definir variables. Las **variables de shell** son variable definidas sólo para la shell que estamos usando. Los programas ejecutados por la shell no tienen dichas variables. Por ejemplo, para definir una variable de shell:

```
mivar=hola
```

El nombre de la variable es *mivar* y su valor la cadena de texto *hola*.

La shell, cuando ve en una línea `$mivar`, sustituye esa cadena por el valor de esa variable. Si no existe la variable, la sustituye por la cadena vacía. Por ejemplo:

```
 $ mivar1=hola
 $ mivar2=pepe
 $ echo mivar1
mivar1
 $ echo $mivar1
hola
 $ echo $mivar1 $mivar2
hola pepe
 $ echo $noexiste
 $
```

Hay otro tipo de variables. Las **variable de entorno** no son sólo para la shell. Estas variables las heredan los procesos creados por esta shell (no por otros). En general, un proceso hereda las variables de entorno de su creador (no importa si es un shell o cualquier otro programa) y tendrá sus propias copias de esas variables, con el mismo valor.

Para hacer que una variable de shell sea una variable de entorno, hay que exportar la variable. Esto también es importante: cada proceso tiene su copia de la variable; si se modifica, no tiene efecto en los otros procesos que la han heredado (pero sí en los que la hereden de este proceso). Para hacer que una variable de la shell pase a ser una variable de entorno, hay que usar el comando `export`:

```
export mivar
```

Se puede dar valor y exportar la variable en un sólo comando:

```
export mivar=bla
```

El comando `set` muestra todas las variables (de shell y de entorno). El comando `printenv` muestra las variables de entorno (también lo hace el comando `env`). El comando `unset` elimina una variable.

Por ejemplo:

```
 $ var1=hola
 $ var2=adios
 $ echo la primera variable es $var1
la primera variable es hola
 $ echo la otra es $var2
la otra es adios
 $ export var1
 $ echo esta variable $noexiste NO existe
esta variable NO existe
 $ unset var2
 $ echo ahora var2 ya no existe: $var2
ahora var2 ya no existe:
 $
```

En un sistema Unix tenemos varias variables populares:

- `$PATH`: la ruta para buscar los ficheros ejecutables para los los programas que se intentan ejecutar en la shell. La variable contiene una lista de rutas separadas por dos puntos (:).
- `$HOME`: la ruta de tu directorio casa.
- `$USER`: el nombre de usuario.
- `$PWD`: la ruta del directorio de trabajo de la shell.
- `$LANG`: configuración de localización (*locales*). Esto indica el idioma, formato de fechas, etc.
- `$LC_xxx`: otras variables de localización (*locales*).

1.4.6. Usando el terminal

Cuando trabajamos en un terminal, podemos usar combinaciones de teclas para realizar algunas acciones:

- ↑ repite los comandos ejecutados anteriormente en la shell.
- `Tab` El tabulador completa nombres de ficheros.
- `Ctrl+r` busca comandos que ejecutamos hace tiempo[16].
- `Ctrl+c` mata el programa que se está ejecutando.
- `Ctrl+z` detiene el programa que se está ejecutando.

[16] Ctrl+r significa presionar a la vez la tecla Ctrl y la tecla r.

- `Ctrl+d` termina la entrada (o manda lo pendiente).
- `Ctrl+a`: mueve el cursor al principio de la línea.
- `Ctrl+e`: mueve el cursor al final de la línea.
- `Ctrl+w`: borra la palabra anterior en la línea.
- `Ctrl+u`: borra desde el cursor hasta el principio de la línea.
- `Ctrl+k`: borra desde el cursor hasta el final de la línea.
- `Ctrl+s`: congela el terminal. `Ctrl+q` lo descongela.
- `Ctrl+l`: limpia el terminal (borra las líneas y deja sólo el prompt).

Cuanto antes te acostumbres a usar estas combinaciones, más tiempo ahorrarás escribiendo comandos en la shell.

Ayuda para la configuración

El comportamiento de algunas partes del sistema (por ejemplo, la salida de algunos comandos o el procesado de texto con expresiones regulares y *globbing*, que veremos en el siguiente capítulo en la Sección 2.20) depende de las *locales*, es decir, de la configuración del idioma, país, etc. Se recomienda tener la variable de entorno `LANG` puesta a `C`.

Si se quiere configurar de forma no persistente, basta con escribir `export LANG=C` en el terminal. Si se quiere configurar de forma persistente, hay que cambiar un fichero de configuración para la shell interactiva por omisión del sistema. En Ubuntu, la shell por omisión es `bash`. Para configurarlo en bash, se puede ejecutar lo siguiente (copia con cuidado los dos comandos, con las comillas dobles, el igual y los dos mayores):

```
$ cd
$ echo "export LANG=C" >> .bashrc
$
```

En algunas distribuciones, el tabulador escapa el $ en una variable en la shell bash. Esto significa que escribir `$HOME/t` y apretar el tabulador completa `\$HOME/t` en lugar de `/home/`*`tulogin`*`/tmp`.

Los comandos `shopt -s direxpand` o `shopt -s cdable_vars` (producen comportamientos levemente diferentes, pruébalos) desactivan este comportamiento. Si quieres que sea permanente:

```
$ cd
$ echo "shopt -s direxpand" >> .bashrc
$
```

1.5. Ejercicios no resueltos

1. Instala VirtualBox o QEMU en tu máquina. Instala Linux dentro. †∗ Prueba diferentes configuraciones. Instala otra distribución diferente en la misma máquina virtual. Haz que se puedan arrancar ambas alternativamente.†∗

2. Instala un Linux nativo en tu máquina.† † ∗ Asegúrate de hacer antes una copia de seguridad de tus ficheros copiándolos a un disco externo.

3. Escribe una línea de comando errónea sin ejecutarla (sin darle al *Enter*). Prueba las combinaciones de teclas de 1.4.6. Encuentra la combinación que utiliza menos teclas para corregirla. Haz esto varias veces para diferentes líneas de comandos, algunas que has ejecutado ya, con diferentes errores, etc. †∗

4. Escribe un pequeño programa en tu lenguaje de programación favorito usando `vi`. Configura `vi` para que la tabulación use un sólo carácter tabulador y para que la codificación sea UTF-8. Utiliza el comando `od` sobre un fichero con texto editado por tí desde cero para comprobar que esto es así.†∗

5. Lee las páginas de manual de introducción del sistema:

   `man man`, `man 1 intro`, `man 2 intro`... † ∗ ∗

6. Lee las páginas de manual de los comandos explicados en este capítulo. Intenta familiarizarte con las partes de la página de manual y la sintaxis de los diferentes comandos viendo qué tienen en común.† ∗ ∗

7. Prueba los comandos explicados en este capítulo.† ∗ ∗

8. En una instalación de Linux (nativa o en una máquina virtual) averigua toda la información que puedas sobre la máquina y los drivers. Puedes usar `uname`, `lsmod`, `lsusb`, `lsblk`, `lspci`, `lsscsi`, `lscpu`, `modinfo`, `/proc/meminfo`, `/proc/cpuinfo`... †∗

9. Instala OpenBSD o FreeBSD en VirtualBox o QEMU. Prueba comandos que conozcas en Linux, para ver cuales funcionan y cuales no.† † ∗

10. Instala Docker en tu máquina Linux. Instala varias imágenes y pruébalas.† † ∗

2 Shell

2.1. ¿Qué es la shell?

La *shell* o *intérprete de comandos* (*CLI, **C**ommand **L**ine **I**nterface*) ha sido tradicionalmente la interfaz principal de usuario de un sistema hasta el advenimiento de los interfaces gráficos (*GUI, **G**raphical **U**ser **I**nterface*). Para un recuento histórico de este desarrollo, es recomendable leer el idiosincrático libro de Stephenson [6].

El intérprete de comandos es un programa[1] que lee líneas de texto que son concatenaciones de comandos[2] que ordenan al sistema que realice una tarea. Aunque hoy en día casi todos los sistemas vienen con una interfaz gráfica, sigue siendo importante aprender la shell, a pesar de resultar más complicada. ¿Por qué? El lenguaje es un instrumento poderoso. Realizar tareas mecánicas, repetitivas y complicadas de forma rápida con un ordenador requiere un uso sofisticado del lenguaje. De otra manera nos vemos obligados a realizar estas tareas repetitivas nosotros, perdiendo nuestro tiempo, atención y, sobre todo, haciendo desagradable el uso del ordenador. Si alguna vez has tenido que renombrar, borrar o mover muchos ficheros a mano mediante la interfaz gráfica, has experimentado la desazón que acompaña a estas tareas. ¡Estás realizando el trabajo de la máquina!

El nombre de shell (concha o cáscara) viene de la relación con el sistema, al que envuelve para realizar de interfaz con el usuario (igual que hace el sistema de ventanas, el *GUI* más común en un *PC*). Una representación artística tradicional de esto se puede ver en la figura 2.1 y una algo más precisa se puede ver en la figura 2.2.

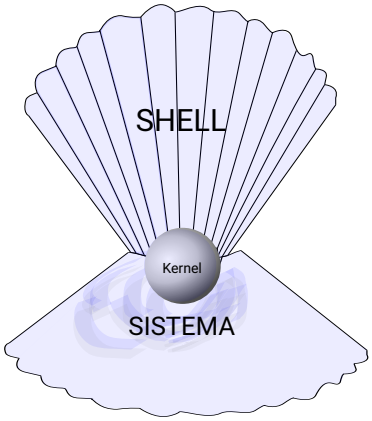

Figura 2.1: Explicación del nombre shell

Para poder interactuar con la shell en un sistema Linux, lo más fácil es abrir un terminal. Un terminal es un programa que controla una ventana y ejecuta una shell, como se puede ver en 2.3. En él podemos escribir comandos. La shell se encargará de leerlos y ejecutarlos.

[1] Otro nombre que recibe es *REPL* (***R**ead **E**valuate **P**rint **L**oop*), por el bucle con el que se construyen los intérpretes de comandos, del que hablaremos y que se ve en la figura 2.4.

[2] Una traducción quizás más precisa sería orden o mandato. Sin embargo la mala traducción *comando* se ha utilizado de forma común y es ya un tecnicismo enquistado entre los programadores.

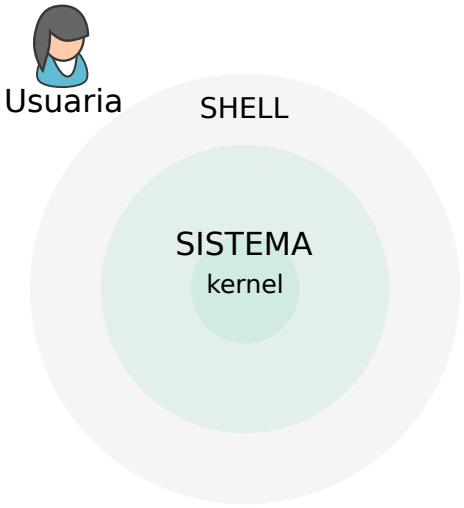

Figura 2.2: Relación más precisa de la shell con el sistema

Cuando la shell ejecuta de este modo, recibe el nombre de shell interactiva. Cuando está lista para leer una línea, escribe una cadena de texto para avisar al usuario. Esta cadena recibe el nombre de *prompt*. En la figura 2.3 el *prompt* que escribe la shell es `paurea@warp:~$`. En ella se puede ver la ejecución de varios comandos.

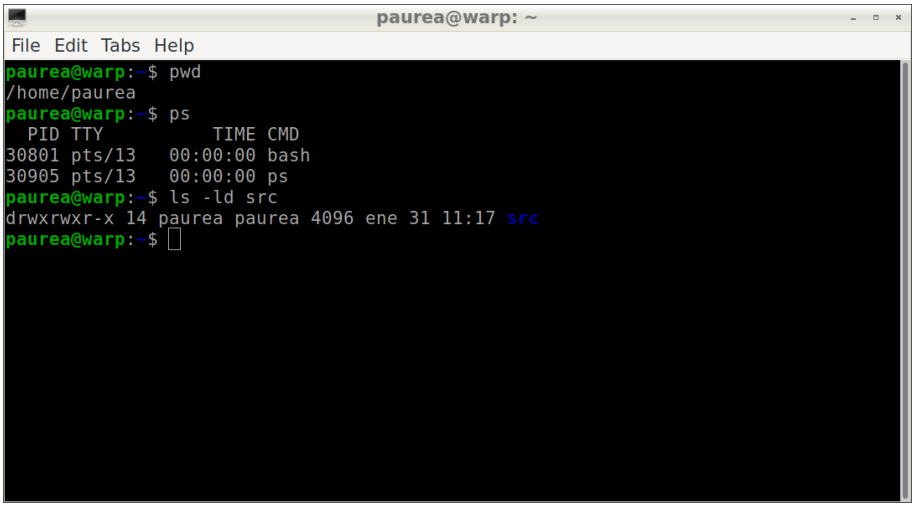

Figura 2.3: Un terminal con una shell y varios comandos

Cuando una shell ejecuta de forma interactiva, su programa principal es un bucle como el que se ve en la figura 2.4. Algunos pasos del bucle no son obligatorios (por ejemplo, esperar a que acaben los comandos al final del diagrama). Tener claros estos pasos nos ayudará a entender qué sucede más adelante.

Una buena introducción al sistema Unix, desde el punto de vista de un usuario pro-

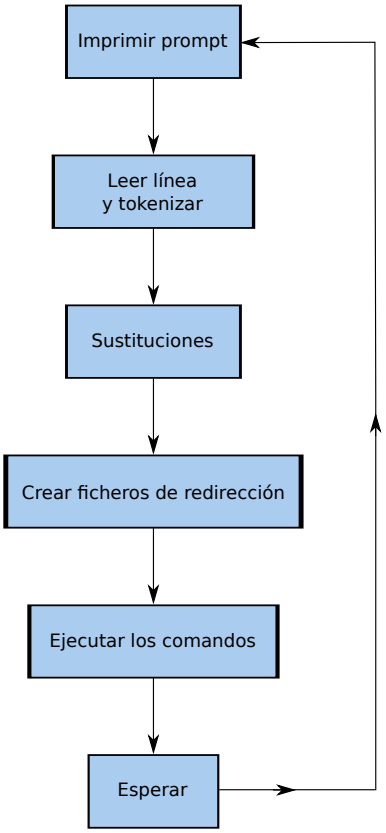

Figura 2.4: Pasos que realiza una shell interactiva

gramador, es el libro clásico de Kernighan y Pike [7]. Este libro puede darnos una idea de la filosofía original del Unix original, que consiste en combinar pequeñas utilidades mediante la shell. Una versión más avanzada y moderna de este libro podría ser el libro de Powers et al. [8], aunque es un monstruo de más de 1000 páginas en el que se pierde un poco el espíritu original de simplicidad de Unix.

Mientras se aprende shell, es muy buena idea tener a mano un ordenador con un sistema de tipo Unix para poder abrir un terminal y probar los comandos de forma interactiva (y ver sus páginas de manual). **Te animamos a que lo hagas**, es la forma de estudiar esta materia. En general, los comandos que se explican aquí deberían funcionar en cualquier terminal de Unix sin problemas, pero recomendamos usar un sistema GNU/Linux para evitar diferencias (opciones de algunos comandos, etc.).

2.2. Comandos y argumentos

En el ejemplo que hemos visto de la figura 2.3 se ejecutan varias líneas de comando sencillas. Una es el comando `pwd`. Como vimos en la introducción, este comando imprime el directorio de trabajo de la shell y no recibe argumentos. Cuando un comando

necesita nombres de ficheros para trabajar o datos de algún tipo se le pueden pasar argumentos que los especifiquen. Los argumentos se escriben después del nombre del comando separados por espacios. En el ejemplo anterior, el comando `ls -ld src` recibe dos argumentos `-ld` y `src`. El primer argumento es especial, porque empieza por `-`. Los argumentos que comienzan por guión se utilizan para especificar *flags* o modificadores. Normalmente, los modificadores se pueden agrupar. Por ejemplo, se podría haber ejecutado `ls -ld src` o `ls -l -d src`. En ambos casos, el comando recibe dos flags, `l` y `d`. Los modificadores sirven para cambiar el comportamiento de un comando y se pueden consultar en la página de manual del comando, por ejemplo, `man 1 ls`[3].

Algunos comandos tienen modificadores que no empiezan por guión (incluso puede que el mismo carácter con guión y sin guión que significan cosas diferentes). Esto es debido a que algunos comandos heredan esta sintaxis sin guión de `BSD`. El caso más famoso es `ps`: el comando `ps -a` no hace lo mismo que el comando `ps a`.

Dos modificadores que aparecen de forma muy común en muchos comandos son `-h` y `-v`. El modificador `-h` sirve para que un comando imprima la ayuda (*help*). En muchos comandos, este modificador también se puede escribir como `--help`. El modificador `-v` (*verbose*) suele modificar cómo de ruidoso o dicharachero es el comando, es decir, cuánta información escribe sobre lo que hace. En cualquier caso, siempre es conveniente leer la página de manual del comando.

A veces, necesitamos que un argumento que no es un modificador comience por `-`. Por ejemplo, si queremos borrar un fichero de nombre `-patata`, le pasamos como argumento anterior al nombre del fichero una flag `--`. Los argumento que siguen a `--` no se interpretarán como modificadores. Por ejemplo `rm -- -patata` borra el fichero `-patata`.

2.3. Shell interactiva y scripts

Hay dos formas de ejecutar la shell. Una es, como hemos visto antes, de forma interactiva, escribiendo líneas de comando que se ejecutan en el momento. La otra es en modo lote o *batch*. Este segundo modo es la forma en la que la shell ejecuta cuando está ejecutando un script. Un script es un programa escrito en shell. Cuando la shell ejecuta en modo *batch*, se comporta de forma diferente. Por ejemplo, no escribe el prompt.[4]

Un fichero en Unix es un programa interpretado mediante un mecanismo basado en lo que se llama el *número mágico* (*magic number*). Cuando se ejecuta un programa, la llamada al sistema `exec` (que estudiaremos en profundidad en el Capítulo 3) recibe la ruta del fichero a ejecutar. Los primeros bytes de dicho fichero se utilizarán para decidir qué se va a hacer en la ejecución. Si los dos primeros bytes son `#!` (sostenido seguido de admiración, que suele recibir el nombre en Unix de *hash bang*), es decir, los bytes $0x21$ $0x23$, se considera que el fichero es un programa interpretado y se ejecutará el intérprete

[3] Es importante recordar que la sección 1 de la página de manual es la que se refiere a comandos de shell. Nos referiremos a un comando en esta página de manual poniendo la sección entre paréntesis *ls(1)*.

[4] La shell realiza varias comprobaciones para saber si tiene que ejecutar en un modo o en el otro. Una de ellas, por ejemplo, es mirar si su entrada y salida estándar (el descriptor de fichero 1 y 2) están conectados a un terminal mediante la función de biblioteca de C `isatty(3)`.

que viene a continuación con sus argumentos (el resto de la línea), pasando como último argumento el la ruta del fichero.

Esto no es especial de la shell, sino que sucede con cualquier programa y lo aprovechan otros intérpretes como Python. Ésta es la razón de que el carácter `#` sea el comentario en estos lenguajes de programación (incluido en la shell)][5]. Así ignoran la primera línea sin tener que hacer nada especial. Si creamos un fichero que comience por esos caracteres y le damos permiso de ejecución, podemos hacer un script con cualquier programa, no sólo con la shell.

Veamos un ejemplo de script de shell:

Script 2.1: script

```
1 #!/bin/sh
2 echo hola
```

Ahora vamos a ejecutarlo. Como ya se explicó en la introducción, el programa `cat` escribe el contenido de los ficheros que recibe como argumento por la salida. El programa `echo` escribe sus parámetros por la salida y `ls` lista los ficheros. El comando `chmod` sirve para dar permisos (en este caso de ejecución) al programa. Veremos más detalles sobre los permisos de los ficheros en el Capítulo 4:

```
 $ cat script
#!/bin/sh
echo hola
 $ chmod +x script
 $ ls -l
total 4
-rwxrwxr-x 1 paurea paurea 52 mar 24 11:09 script
 $ ./script
hola
 $
```

Veamos otro ejemplo. Vamos a ejecutar `cat` como si fuera un intérprete:

Script 2.2: quine

```
1 #!/bin/cat
2
```

[5] Aunque en la mayoría de las shells de Unix se pueda ejecutar un fichero con comandos sin el *hash bang* por una decisión de diseño, no debemos omitirlo de nuestros scripts: siempre tenemos que comenzarlos con `#!`.

```
 $ cat quine
#!/bin/cat

 $ chmod +x quine
 $ ./quine
#!/bin/cat
 $
```

¿Qué ha pasado? Al ejecutar el fichero interpretado, se ha ejecutado `cat` (el intérprete en este caso) pasándole como único argumento la ruta del fichero `quine`. El programa `cat` ha hecho su trabajo: escribir por su salida el contenido del fichero que se le ha pasado como argumento.

Ahora vamos a ejecutar `echo` como intérprete. Se puede ver el paso de argumentos:

Script 2.3: dime

```
1 #!/bin/echo el fichero es
2
```

```
 $ cat dime
#!/bin/echo el fichero es

 $ chmod +x dime
 $ ./dime argumento
el fichero es ./dime argumento
 $
```

Todo esto al final, significa que hay dos maneras de utilizar la shell para ejecutar comandos. Una es escribirlos de forma interactiva en un terminal. La otra es escribir scripts de shell.

Un script es un programa interpretado que utiliza la shell como intérprete. Para depurar estos programas se puede poner `-x` como parámetro para que la shell escriba cada comando del script antes de ejecutarlo:

Script 2.4: hola.sh

```
1 #!/bin/sh -x
2 echo hola mundo
3 pwd
4 exit 0
```

En un script se pueden poner líneas con `echo` para trazar el programa (para depurar como con cualquier otro tipo de programa). Aparte de esto, una de las ventajas de los intérpretes como la shell es que, mientras se escribe el programa, se pueden probar los comandos sueltos de forma interactiva en un terminal.

Cuando tengo que realizar una tarea, ¿cómo decido si utilizar un lenguaje de programación (C, Java, Go, C++, etc.), escribir un script de shell o simplemente ejecutar comandos de forma interactiva? Dependerá mucho del tipo de tarea, su complejidad, la necesidad de automatización (cuantas veces voy a realizarla), etc.

En una primera aproximación, se pueden seguir lo siguientes pasos:

1. Mirar si hay alguna herramienta que haga lo que queremos, es decir, leer la documentación de diferentes herramientas y buscar en el manual.

2. Si no encontramos una que se adapte, se puede intentar combinar distintas herramientas en la línea de comandos. La primera aproximación es una combinación de comandos utilizando la shell de forma interactiva (como veremos más adelante, *pipelines* de filtros, variables, etc.).

3. Si es muy complicado, se puede intentar combinar todavía más herramientas distintas programando un script de shell. La idea principal es combinar herramientas que hacen bien una única tarea para llevar a cabo tareas más complejas.

4. Si no podemos, desarrollamos nuestra propia herramienta programada en C, Python, Java, Ada, Go, etc.

La shell es especialmente buena para tareas que realizo una vez, para tareas que se repiten mucho y hay que automatizar (combinado con un IDE, Makefile o gradle o equivalente) y para procesar texto.

Automatizar una tarea es una inversión de tiempo y esfuerzo mental. El retorno de esa inversión dependerá de cuánto se tarda en ejecutar la tarea manualmente, cuántas veces se realiza y del esfuerzo mental necesario para la automatización. Para realizar este análisis, hay que tener en cuenta el tiempo necesario para aprender una herramienta nueva, el sufrimiento de realizar a mano una tarea repetitiva, etc.

La tarea que más repite un programador es el bucle compuesto por edición, compilación, prueba y depuración. La automatización de cualquier tarea en ese bucle es extremadamente valiosa, puesto que el retorno de inversión será altísimo por el multiplicador de número de repeticiones. Al principio, las herramientas potentes (como la shell) requieren mucho tiempo y un gran esfuerzo de aprendizaje, pero luego lo repagan al ser herramientas de propósito general. Existe también un equilibrio entre automatizar la tarea actual (fácil) y automatizar cualquier tarea general del mismo tipo que podamos encontrarnos en el futuro (complicado acertar, peor retorno a la inversión).

Como en cualquier programa, los principios a la hora de construir código son **simplicidad**, **claridad** y **generalidad**, en ese orden [9].

2.4. Esperar a que acabe el comando

Tanto en una shell interactiva como en un script, la shell ejecuta una línea de comando tras otro. El usuario puede decidir si quiere que la shell espere a que acabe la línea

anterior antes de ejecutar la siguiente[6]. Si el comando acaba en punto y coma ';' o final de línea, la shell esperará a que acabe antes de ejecutar el siguiente. Si acaba en el carácter *ampersand* (`&`) no esperará.

Si la shell espera, se dice que el comando ejecuta en *primer plano* (*foreground*). Si no, se dirá que ejecuta en *segundo plano* (*background*). La variable de shell especial $! contiene el `pid` del último proceso que ejecutó en segundo plano.

Por ejemplo, a continuación ejecutamos `date` (para imprimir la fecha), `sleep` (para esperar 10 segundos) y `date` de nuevo. Estos tres comandos se ejecutan *secuencialmente*: hasta que no termina uno, no comienza la ejecución del siguiente. Después, se ejecutan los mismos comandos pero no se espera a que el `sleep` termine para ejecutar el último `date`:

```
 $ date; sleep 10; date;
sáb abr  4 13:36:34 CEST 2020
sáb abr  4 13:36:44 CEST 2020
 $ date; sleep 10 & date;
sáb abr  4 13:36:47 CEST 2020
[1] 21791
sáb abr  4 13:36:47 CEST 2020
 $
```

En el caso de una shell interactiva, al no esperar, escribirá el *prompt*. Esto puede ser un problema si el programa ejecutando en *background* escribe algo por su salida (se puede mezclar el texto de salida y el prompt):

```
 $ echo '
hola
vamos
a escribir
varias lineas' &
[1] 21573
 $
hola
vamos
a escribir
varias lineas
```

2.5. Diferentes shells

Hay muchas shells diferentes. `sh` es la shell original de Unix, escrita por Ken Thompson. Esa shell fue reescrita por Stephen Bourne en 1979 para Unix Version 7: *bourne shell*.

[6] Esto se corresponde con que la shell llame a `wait(2)` para los procesos hijo que derivan de esa línea de comando. Veremos esa llamada al sistema en el Capítulo 3.

Los sistemas derivados de Unix incluyen distintas shells: `sh`, `ash`, `bash`[7], `dash`, `ksh`, `csh`, `tcsh`, `zsh`, `rc`, etc. Cada una tiene sus características, pero también tienen mucho en común. En sistemas modernos, `/bin/sh` suele ser un enlace simbólico[8] a su shell por omisión para ejecutar scripts. La shell por omisión para ejecutar scripts suele mirar el nombre con el que se ejecuta, y si es `sh`, se comporta de forma más estricta. En Ubuntu y Debian, la shell por omisión para ejecutar scripts es `dash`. No hay que confundir la shell para ejecutar scripts con la shell interactiva (para un usuario), que por omisión es `bash` en casi todos los Linux (y algunos otros Unix). Dos buenas referencias para aprender bash son su página de manual *bash(1)* y el libro de Newham y Rosenblatt [10].

Una manera de asegurarse que los scripts pueden ser portables entre distintos sistemas es utilizar `sh` (es decir, que la primera línea del script sea `#!/bin/sh`) y utilizar únicamente las características POSIX (IEEE Std 1003.1-2017). De esta manera, sólo utilizamos el subconjunto común que implementan la mayoría de las shells. Esto es lo que haremos en el libro y lo que recomendamos en general. Los scripts de shell, mejor escribirlos en `sh`, por portabilidad y para que el comportamiento sea predecible. Otro buen libro para aprender a programar scripts de shell es el de Kochan y Wood [11].

2.6. Comandos internos o builtins

Hay comandos que se implementan dentro de la shell: no se ejecuta un fichero externo a la shell, sino que son parte de la propia shell. Se llaman *builtin*. Un ejemplo de esto es el *builtin* `exit`. Su ejecución hace que acabe el script con el estado (*status*) indicado en su argumento. Si un script no sale con `exit`, cuando acaba sale con el estado del último comando que ejecutó (el valor de `$?` como veremos en la sección 2.9).

Script 2.5: hola.sh

```
1 #!/bin/sh
2 echo hola mundo
3 pwd
4 exit 0
```

Si se quiere ver de qué tipo es un comando (builtin, script de shell...) se puede usar `type`.

```
 $ type exit
exit is a shell builtin
 $ type rm
rm is /usr/bin/rm
```

[7] Una guía de estilo interesante para bash que se usa en google se puede encontrar en https://google.github.io/styleguide/shellguide.html

[8] Un enlace simbólico es una referencia a otro fichero. En otros sistemas se llaman *accesos directos*. Veremos más sobre enlaces en el Capítulo 4.

2.7. Sustituciones

Cuando una shell ejecuta, sea de forma interactiva o no (la diferencia con respecto a la figura será que imprima o no el prompt), realiza el conjunto de tareas que se puede ver en la Figura 2.4. El tercer paso será sustituir algunos de los trozos de texto de la entrada por otros.

Para evitar sustituciones, igual que para hacer que cualquier carácter se represente a sí mismo y no tenga ningún valor especial en la shell, se pueden utilizar los llamados *caracteres de escape*. Son tres, la barra invertida o *backslash* que es el caracter '\', las comillas simples 'texto'.[9] o las comillas dobles "texto". La barra invertida antes de un carácter hará que sencillamente se interprete como él mismo \#, por ejemplo, representa el carácter '#' y no comienza un comentario.

De forma similar, las comillas simples (también conocidas como *comillas duras*) escapan todos los caracteres que hay en su interior. Así '#$?' es igual que si hubiesemos escapado los tres caracteres por separado: \#\$\?.

Las comillas dobles, conocidas también como *comillas blandas*, escapan todos los caracteres salvo tres: el dólar (que afecta a la sustitución de las variables, que veremos a continuación), la comilla invertida (que afecta hasta que se cierre) y la barra invertida (que afecta al siguiente carácter para escaparlo).

Por ejemplo, dentro de las dobles comillas, se sustituyen las variables y los resultados de comandos:

```
 $  echo "$a   $(echo xy | wc -c) ‘echo rrr‘ \$"
zzz   3 rrr $
 $
```

2.7.1. Variables

En la introducción ya vimos cómo funcionan las variables en la shell. Ahora profundizaremos un poco en esta cuestión.

Un tipo particular de sustituciones es la sustitución de variables. Un nombre precedido de un dólar (`$nombre`) se sustituirá por el contenido de la respectiva variable de shell. Las variables se definen con el builtin infijo =. Es importante recordar que la sustitución la hace la propia shell tras leer la línea y antes de ejecutar los comandos, como se puede ver en la Figura 2.4.

En el siguiente ejemplo, el comando `echo` no sabe nada sobre $x, es la shell la que sustituye el texto ($xy) por el valor de la variable y ejecuta la línea `echo bla`:

```
 $ x=bla
 $ echo $x
bla
 $
```

[9] Ojo, las comillas simples se corresponden con el carácter que aparece en la tecla a la derecha del cero, la que tiene el interrogante en el teclado español.

En este otro ejemplo vemos como podemos ejecutar un comando (en este caso, el comando `ls`) usando una variable:

```
 $ touch c d e
 $ ls
c  d  e
 $ cmd=ls
 $ $cmd
c  d  e
 $
```

Es importante entender que estas variables se definen en la shell, los *hijos* de la shell (los procesos que crea para ejecutar los comandos) no las heredan. Para que los hijos puedan heredar esas variables, habrá que exportarlas al *entorno*.

El entorno es un trozo de memoria que los procesos heredan de su padre en Unix. Una parte del entorno son las *variables de entorno*, que no deben confundirse con las variables de la shell (esto es, las variables no exportadas). Como ya vimos en la introducción, el comando builtin `export` sirve para meter a una variable de shell en el entorno y que se pase de padres a hijos.

```
 $ hola=bla
 $ echo $hola
bla
 $ bash -c 'echo $hola'

 $ export hola
 $ bash -c 'echo $hola'
bla
 $ export adios=patata
 $ echo $adios
patata
 $ bash -c 'echo $adios'
patata
 $
```

Los programas para Unix, además, pueden consultar y cambiar estas variables aunque no sean shell scripts[10]. Esto es bastante común, es una forma de configurar el sistema.

[10] En el Capítulo 3 veremos cómo hacerlo con funciones como `getenv(3)`.

2.7.2. Sustitución de resultado de comando

Esta sustitución sustituye un comando por lo que éste escribe en su salida. Se puede escribir de dos formas: `$(comando)` y `‘comando‘`.

Por ejemplo, aquí la shell ha sustituido `‘echo hola‘` por `hola`, que es el argumento que recibe el `echo` para imprimir por su salida:

```
 $ echo ‘echo hola‘
hola
 $
```

En el siguiente ejemplo, se crea una variable de shell con la salida del comando `wc -l /tmp/a | cut -d’ ’ -f1`, que cuenta las líneas del fichero `/tmp/a`:

```
 $ vv=$(wc -l /tmp/a | cut -d’ ’ -f1)
 $ echo $vv
31
 $
```

2.8. Redirecciones

2.8.1. Entrada estándar, salida estándar y redirecciones

En Unix, los ficheros abiertos de un proceso están numerados (***F**ile **D**escriptor*, FD o *descriptor de fichero*). Cada vez que un proceso abre un fichero para trabajar con él[11] se le asigna un número en la tabla de descriptores abiertos para ese proceso (cada proceso tiene su tabla) tras comprobar los permisos pertinentes.

Hay tres ficheros que vienen abiertos por omisión: entrada estándar (0, *standard input*, *stdin*), salida estándar (1, *standard output*, *stdout*) y salida de errores (2, *standard error*, *stderr*).

- *stdin* es de donde el programa lee datos por omisión.
- *stdout* es donde escribe datos por omisión.
- *stderr* es donde escribe mensajes de error por omisión.

La salida estándar y la salida de errores están separadas para que no interfieran una con otra (datos de salida vs. notificaciones y errores).

Cuando se crea un nuevo proceso, el proceso creador le deja estos tres ficheros configurados como necesite. Un programa escrito para Unix puede contar con que esos tres ficheros están configurados y listos para ser usados. Este es uno de los convenios más importante en los sistemas de tipo Unix.

[11] Se hace con la llamada *open(3)* sobre una ruta para abrir un fichero, lo veremos en el Capítulo 4.

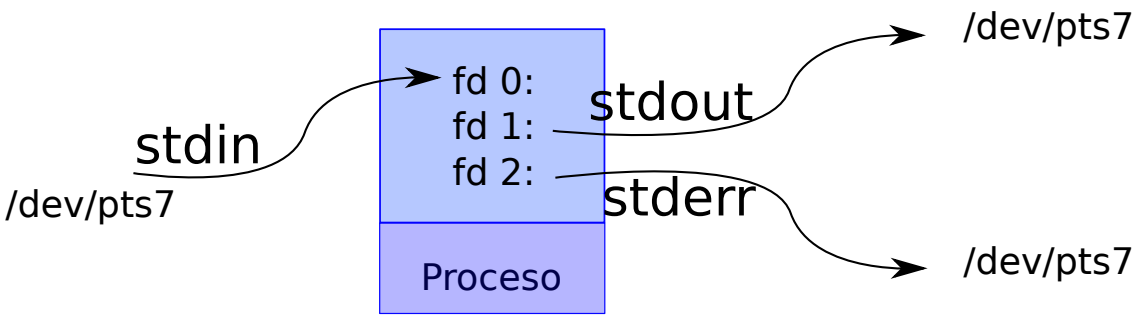

Figura 2.5: Entradas y salidas de un proceso

En una shell interactiva, por lo general (si no se realiza ninguna *redirección*), esos tres primeros descriptores de fichero están configurados para acceder a dispositivos en los que leer o escribir interactúa con la *consola* (es decir, escribir imprime algo por el terminal y leer obtiene datos del usuario por el teclado). En Unix, el terminal está representando por un fichero en `/dev/`. Por ejemplo, en la Figura 2.5, el terminal que usa ese shell es `/dev/pts7`. Cuando se lee de ese fichero, se lee del teclado. Cuando se escribe en ese fichero, se imprimen caracteres en la pantalla.

Estos descriptores de fichero se pueden redireccionar a otros ficheros mediante la shell antes de ejecutar el proceso. De esta forma, podemos guardar la salida de un programa, dársela a otro programa, etc. Los detalles de cómo funciona la redirección en la shell se puede ver en *dash(1)* (ver la sección *Redirections* de la página de manual).

Para redireccionar la entrada y la salida estándar de un comando, se utilizan los caracteres mayor y menor, '<' y '>'. Estos caracteres hacen que el proceso creado por la shell abra los ficheros de redirección antes de ejecutar el comando. En el caso del fichero de salida, lo crea si no existe y lo trunca si ya existe (es decir, lo deja vacío).

Por ejemplo, si ejecutamos los siguientes comandos:

```
 $ ls /tmp/fich
ls: cannot access '/tmp/fich': No such file or directory
 $ echo hola > /tmp/fich
 $ cat /tmp/fich
hola
 $ echo adios > /tmp/fich
 $ cat /tmp/fich
adios
  $
```

Ese ejemplo ejecuta `echo` con la salida en `/tmp/fich` como se ve en la figura 2.6.

Para redireccionar la salida de errores se usa '2>'. En general, poniendo un número antes del símbolo de redirección, se provoca que se redireccione el descriptor indicado por dicho número (p. ej. '5<' indica que se use el fichero indicado para redireccionar el descriptor 5). Recuerda que sólo los tres primeros (0,1 y 2) son especiales.

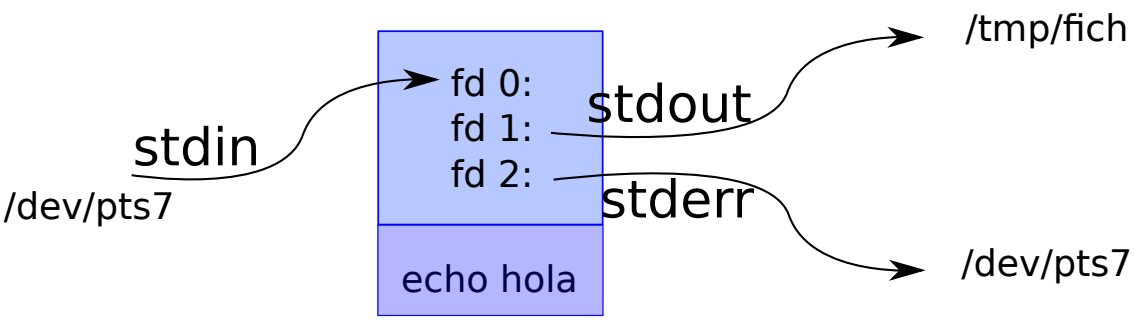

Figura 2.6: Ejecución con la salida estándar apuntando a fich

Por ejemplo, el siguiente ejemplo ejecuta `ls` con la salida de errores redirigida al fichero `errores`. Por tanto, cualquier error que escriba `ls` acabará en dicho fichero y no lo veremos en el terminal:

```
 $ ls /noexiste 2> errores
 $ cat errores
ls: cannot access '/noexiste': No such file or directory
 $
```

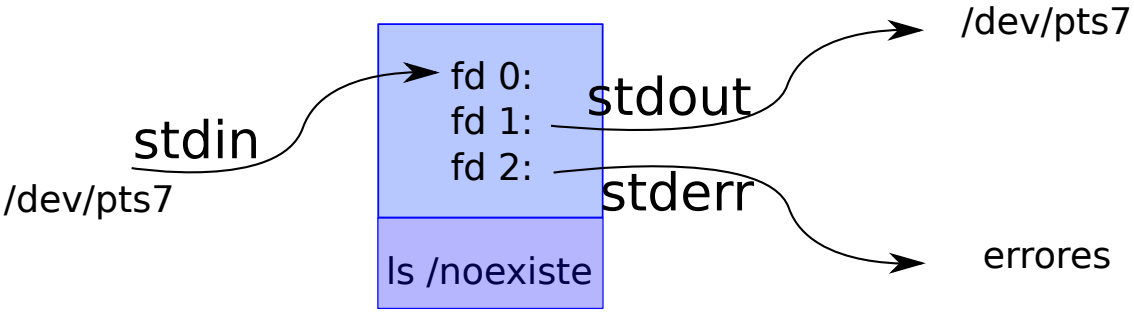

Figura 2.7: Uso de la salida de error

Si no queremos que se trunque el fichero que ponemos a la salida, '≫' hace lo mismo que '>' pero abre el fichero en modo *append* (añadir). Esto significa que los datos se añaden al final en lugar de hacer como '>', que trunca el fichero.

```
 $ ls /tmp/fich
ls: cannot access '/tmp/fich': No such file or directory
 $ echo hola > /tmp/fich
 $ cat /tmp/fich
hola
 $ echo adios >> /tmp/fich
 $ cat /tmp/fich
hola
adios
 $
```

¡Atención! Un error muy común es ejecutar un comando y usar para la redirección de salida el mismo fichero que se pasa como argumento para leer:

```
 $ echo aaaaaaa > bla
 $ cat bla
aaaaaaa
 $ cat bla > bla
 $ cat bla
 $
```

Esto deja el fichero vacío porque el fichero se trunca *antes de ejecutar el comando*. Basta con ver la figura 2.4. Lo mismo puede pasar haciendo una redirección de entrada y otra de salida con el mismo fichero:

```
 $ echo hola > bla
 $ wc -l < bla > bla
 $ cat bla
0
 $
```

Antes de ejecutar `wc`, el fichero `bla` tiene una línea, pero al ejecutar `wc` cuenta cero líneas (porque la redirección de salida lo ha truncado).

Para mandar la salida de error al mismo sitio que la estándar o viceversa se puede poner $n>\&m$ donde n y m son dos números de descriptores de fichero.

Hay que tener cuidado: el orden importa y se interpreta de izquierda a derecha. Esto redirecciona al fichero `fich` la salida estándar. Luego, redirecciona la salida de errores al mismo sitio que la estándar, es decir, también a `fich`, como se ve en la figura 2.8:

```
 $ ls /noexiste > fich 2>&1
 $ cat fich
ls: cannot access '/noexiste': No such file or directory
 $
```

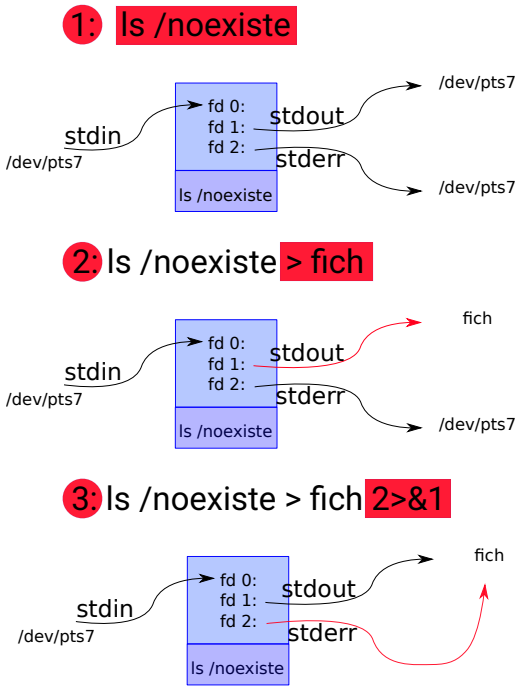

Figura 2.8: Redirección al fichero `fich` de la salida estándar y la salida de errores, ilustrado en tres pasos.

Sin embargo, esto es diferente de lo anterior y deja el fichero vacío, como se ve en la figura 2.9.

```
 $ ls /noexiste 2>&1 > fich
ls: cannot access '/noexiste': No such file or directory
 $ cat fich
 $
```

Para escribir algo por la salida de error desde un script, por ejemplo con `echo`, tenemos que hacer que la salida estándar de ese `echo` sea el fichero que tiene el script abierto en la salida de errores:

```
 echo aaa 1>&2
```

Así el script:

Script 2.6: aaaerror.sh

```
1 #!/bin/sh
2 echo aaa 1>&2
```

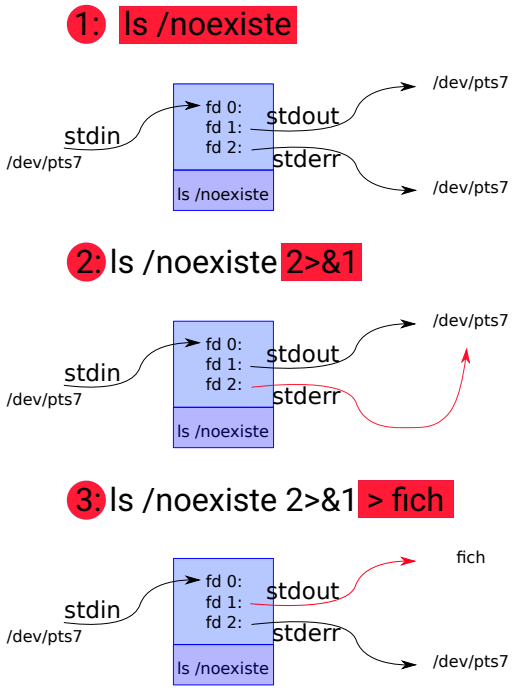

Figura 2.9: Redirección incorrecta al fichero `fich` de la salida de error, ilustrado en tres pasos.

Escribe por su salida de error:

```
 $ ./aaaerror.sh 2> /tmp/err
 $ cat /tmp/err
aaa
 $
```

2.8.2. Ficheros especiales

Hay una serie de ficheros especiales que ofrece el kernel para interactuar con él. Se llaman ficheros sintéticos (no hay ningún fichero en disco real que se corresponda con ellos). Un ejemplo de sistema de ficheros sintético es `/proc` (ver `proc(5)`). En ese directorio todos los ficheros y directorios son sintéticos y sirven para dar información del sistema y de sus proceso. Otro ejemplo es `/dev/pts3` que representa una consola.

El sistema de ficheros `/proc` se puede utilizar para pedir información sobre los ficheros que tiene abierto un proceso. Por ejemplo para ver los ficheros que tiene abiertos la shell se pueden utilizar los siguientes comandos:

```
 $ echo $$
14927
 $ ps
  PID TTY          TIME CMD
 14927 pts/9   00:00:00 bash
 14928 pts/9   00:00:00 ps
 $ ls -la /proc/$$/fd
total 0
dr-x------ 2 paurea paurea  0 feb 11 15:45 .
dr-xr-xr-x 9 paurea paurea  0 feb 11 15:45 ..
lrwx------ 1 paurea paurea 64 feb 11 15:45 0 -> /dev/pts/9
lrwx------ 1 paurea paurea 64 feb 11 15:45 1 -> /dev/pts/9
lrwx------ 1 paurea paurea 64 feb 11 15:45 2 -> /dev/pts/9
lrwx------ 1 paurea paurea 64 feb 11 15:49 255 -> /dev/pts/9
 $
```

La variable de shell $$ contiene el *PID* o número identificador de proceso de la shell en ejecución. En el Capítulo 3 hablaremos más sobre `/proc`.

Otros dos ejemplos de ficheros sintéticos muy útiles a la hora de utilizar la shell son `/dev/null` y `/dev/zero`.

El fichero `/dev/null` es un *agujero negro*, un sumidero en el que todo lo que escriba se pierde para siempre. En ocasiones también se lee de `/dev/null`, cualquier lectura retornará *EOF*(***E**nd **O**f **F**ile*): es como si estuviese vacío. Por ejemplo, es común pasarle a un comando que busca cosas (como `grep`) `/dev/null` como argumento adicional. De esta manera (al recibir más de un fichero) escribirá el nombre del fichero en el que encuentra cosas. En `/dev/null` no encontrará nada, porque parece estar vacío.

Por ejemplo, si ejecutamos este comando, no escribe nada por la salida porque lo escribe todo en `/dev/null`.

```
 $ wc -l < /etc/os-release > /dev/null
```

Un diagrama de este proceso en ejecución se puede ver en la figura 2.10.

Hacer callar a un proceso cuando nos interesa es bastante útil. A veces, en un script, se ejecutan comandos para ver el estado de salida, por ejemplo, comparar dos ficheros con `cmp`. Como no me interesa su salida, la redirecciono a `/dev/null` y así la silencio.

El fichero `/dev/zero` es una *fuente* de bytes a cero, cualquier lectura retornará bytes a cero. Por ejemplo, este comando genera un fichero con 1kB de ceros:

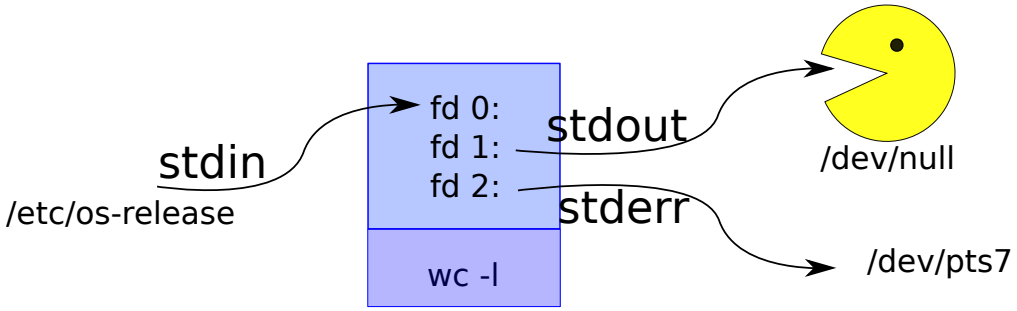

Figura 2.10: Hacer callar a un proceso mediante `/dev/null`

```
 $ dd if=/dev/zero bs=1K count=1 of=/tmp/x
1+0 records in
1+0 records out
1024 bytes (1,0 kB, 1,0 KiB) copied, 0,000134555 s, 7,6 MB/s
 $ ls -l /tmp/x
-rw-rw-r-- 1 paurea paurea 1024 abr  4 19:15 /tmp/x
 $
```

El comando `dd` es útil para crear ficheros con un tamaño definido. Se le puede decir de cuánto en cuánto copia (esto es, el tamaño de bloque que usa), fichero fuente, fichero destino, etc. Como siempre, tienes toda la información relevante en *dd(1)*.

Un truco avanzado para crear un fichero de 10MB extremadamente rápido (y si tenemos suerte, el fichero sólo ocupa un byte[12].) es éste:

```
 $ dd if=/dev/zero bs=1 count=1 seek=10485759 of=/tmp/xx
1+0 records in
1+0 records out
1 byte copied, 9,769e-05 s, 10,2 kB/s
 $ ls -lh /tmp/xx
-rw-rw-r-- 1 paurea paurea 10M abr  4 19:27 /tmp/xx
 $ ls -l /tmp/xx
-rw-rw-r-- 1 paurea paurea 10485760 abr  4 19:27 /tmp/xx
 $
```

Un comando relacionado para partir un fichero en trozos de igual tamaño (o en N trozos) es `split`. Otro más sofisticado para tratar con ficheros de texto (partir por tamaño o por expresiones regulares del contenido) es `csplit`.

[12]Si el sistema soporta ficheros *sparse* o con agujeros *ficheros con huecos*, ver 4.3.7 para más detalles.

2.8.3. Pipes

Un pipe es un mecanismo que sirve para conectar dos descriptores de fichero. Por ejemplo, para conectar la salida (*stdout* o *stderr*) de un proceso con la entrada de otro (*stdin*). Una línea de comando no tiene por qué ser un sólo comando, pueden ser varios conectados por pipes: un *pipeline*. De ahí aparece el concepto de comando filtro. Un filtro lee de la entrada, procesa texto, escribe en su salida.

El siguiente ejemplo cuenta las entradas del directorio `/tmp`. El comando `ls` lista las entradas de ese directorio (una por línea) y `wc -l` cuenta las líneas de su entrada (que es la salida de `ls`):

```
 $ ls -la /tmp | wc -l
40
 $
```

Los procesos en ejecución se muestran en la figura 2.11.

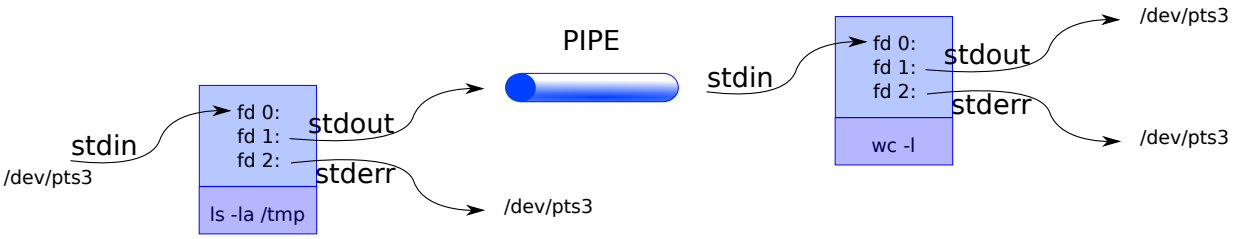

Figura 2.11: Conexión de dos procesos mediante pipes.

Otro ejemplo un poco más avanzado que escribe en el terminal 35 caracteres 'z' podría ser el siguiente:

```
 $ dd </dev/zero bs=35 count=1 2> /dev/null |tr '\0' z
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz$
```

Lo que se ha hecho en ese ejemplo es usar `dd` para leer 35 bytes a cero (`0x00`, el *carácter nulo*) de `/dev/zero` y después usar el filtro `tr` para traducir todos los caracteres nulos en caracteres 'z'. Así conseguimos 35 caracteres 'z'.

En ocasiones es interesante ver el resultado de un programa y escribirlo en un fichero simultáneamente. Esto es lo que hace el comando `tee` (ver Figura 2.12).

```
 $ echo 'hola
  que tal' | tee fich
hola
que tal
 $ cat fich
hola
que tal
 $
```

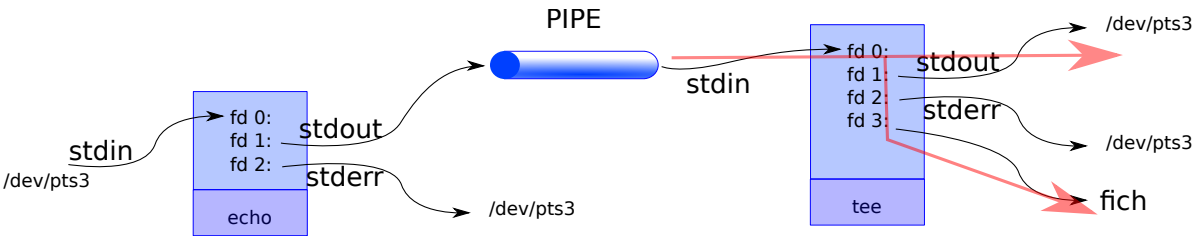

Figura 2.12: Ejemplo de uso del comando tee

Este comando se puede utilizar en medio de un pipeline para espiar los datos que pasan por un punto:

```
 $ echo 'hola
  adios' | tee fich | wc -l
2
 $ cat fich
hola
adios
 $
```

2.9. Estado de salida (status)

Cuando un comando sale, le notifica su estado de salida al proceso padre de salida (*status*)[13]. El estado del último comando (o *pipeline*) que se ejecutó se guarda en la variable de shell `$?`.

Es un número. Si ese número es 0, el comando ha tenido éxito. En otro caso, ha habido un error. Todos los programas tienen que acabar comunicado su estado de salida a su creador. Los scripts de shell no son una excepción: se puede (y se debe) terminar el script notificando el estado de salida mediante el comando builtin `exit`, que recibe un número como parámetro. De esta forma, el script informa al sistema (por ejemplo a otro script o a la shell que lo ejecutó) sobre si ha habido un error en su ejecución o todo ha salido bien.

La *ejecución condicional* de la shell está construida alrededor del estado de salida de los comandos, que se utilizan como valores booleanos. True se representa mediante el estado de salida de éxito (estado 0) y false mediante el de error (estado distinto de 0). Por eso hay dos comandos, `true` y `false`, cuyo única funcion es salir con éxito o fallo. También hay dos builtins infijos, && y ||, que se pueden utilizar como operadores lógicos.

[13] El estado de salida se notifica en el argumento que se pasa a la llamada al sistema `exit`. Si hay varios tipos de errores, pueden devolverse diferentes valores de `exit` (aunque todos diferentes de 0). Todo esto lo veremos más en profundidad en 3.4.1.

Se utilizan también por su propiedad de *cortocircuito*[14]: la línea de comando `cmd1 && cmd2` ejecuta el segundo comando (`cmd2`) sólo si el primero (`cmd1`) ha tenido éxito (por tanto, se hace cortocircuito). Como consecuencia, la línea de comando `cmd1 && cmd2` sólo tiene éxito si ambas líneas tienen éxito (*and* del álgebra de Bool). De forma similar, `cmd1 || cmd2` ejecuta el segundo comando (`cmd2`) sólo si el primero (`cmd1`) ha fallado. Algunos ejemplos se pueden ver a continuación:

```
 $ true
 $ echo $?
0
 $ false
 $ echo $?
1
 $ true && echo hola
hola
 $ false && echo hola
 $ true || echo hola
 $ false || echo hola
hola
 $
```

El comando `true` se podría escribir como un script (no se suele hacer así por eficiencia):

Script 2.7: true

```
1 #!/bin/sh
2 exit 0
```

Lo mismo con el comando `false`:

Script 2.8: false

```
1 #!/bin/sh
2 exit 1
```

2.9.1. Test

El comando `test` sirve para comprobar condiciones de distinto tipo. El comando termina con un estado de éxito o de fallo dependiendo de la condición. Por ejemplo, para ver si existe un fichero se usa la opción `-f` seguida de una ruta:

[14] También conocido como evaluación de McCarthy, el cortocircuito se garantiza en muchos lenguajes de programación (p. ej. el operador `&&` en C), pero en otros lenguajes no (p. ej. en ADA el operador `and` normal no tiene cortocircuito, pero el operador `and then` sí).

```
 $ ls -l x
-rw-rw-r-- 1 paurea paurea 0 mar 27 09:29 x
 $ test -f x
 $ echo $?
0
 $ rm x
 $ test -f x
 $ echo $?
1
 $
```

Algunas de las comprobaciones que puede hacer `test` son:

- Condiciones sobre ficheros:
  - `-f ruta`: si existe y es un fichero.
  - `-d ruta`: si existe y es un directorio.
- Condiciones sobre cadenas:
  - `-n cadena1`: si la longitud de la cadena no es cero.
  - `-z cadena1`: si la longitud de la cadena es cero.
  - `cadena1 = cadena2`: si son iguales.
  - `cadena1 != cadena2`: si cadena1 y cadena2 no son idénticas.
  - `cadena1`: si la cadena no es nula.
- Condiciones sobre enteros:
  - `entero1 -eq entero2`: si los enteros entero1 y entero son iguales.
  - `entero1 -ne entero2`: si los enteros entero1 y entero son distintos.
  - `entero1 -gt entero2`: si entero1 es mayor que entero2.
  - `entero1 -ge entero2`: si entero1 es mayor o igual que entero2.
  - `entero1 -lt entero2`: si entero1 es menor que entero2.
  - `entero1 -le entero2`: si entero1 es menor o igual que entero2.

Hay dos comando `test` instalados en el sistema: el propio `test` y otro comando llamado `[`. El comando `[` es un comando normal, que sencillamente tiene un nombre peculiar y que espera que su último argumento sea `"]"`. De esta forma los `if` y demás estructuras de control de la shell que veremos a continuación pueden tener un aspecto normal.

El comando `[ $a -eq $b ]` hace lo mismo que el comando `test $a -eq $b`. De hecho, en mucho sistemas, `test` mira el nombre con el que se le ha llamado y en el caso de que su nombre sea "[", mira que el último argumento sea "]".

```
 $ which [
/usr/bin/[
 $ touch /tmp/x
 $ [ -f /tmp/x ]
 $ echo $?
0
 $ test -f /tmp/x
 $ echo $?
0
 $ [ 0 -lt 10 ]
 $ echo $?
0
 $ test 0 -lt 10
 $ echo $?
0
 $
```

El comando `which` se encarga de decir la ruta del fichero que se ejecutará cuando intentemos ejecutar un comando (mira en los directorios de la variable de entorno `PATH`). No siempre nos dice qué se va a ejecutar, porque en la shell puede haber un *builtin* con el mismo nombre[15]. Para saber qué es lo que se va a ejecutar realmente y de qué tipo es, ya vimos el builtin `type`. Así vemos que, en las shells modernas, `test` suele ser un builtin:

```
 $ type [
[ is a shell builtin
 $ type test
test is a shell builtin
 $ type ls
ls is /bin/ls
 $
```

Si queremos ejecutar un binario concreto en lugar de un builtin con el mismo nombre, siempre podemos utilizar la ruta absoluta:

```
 $ which echo
/usr/bin/echo
 $ type echo
echo is a shell builtin
 $ echo esto lo escribe el builtin
esto lo escribe el builtin
 $ /usr/bin/echo esto no lo escribe el builtin
esto no lo escribe el builtin
 $
```

[15]También pueden pasar otras cosas relacionadas con `alias` y funciones, que veremos más adelante.

2.10. Parámetros posicionales

Un script puede acceder a los parámetros posicionales que se han pasado al ejecutarlo. Se pueden acceder mediante las variables especiales de shell $1, $2, $3... La variable $0 se sustituye por el nombre con el que se ha invocado el script. La variable $# contiene el número de parámetros (sin contar el 0, es decir, el nombre con el que se llamó al script).

La variable $* contiene todos los parámetros posicionales. La variable "$*" es decir, si ponemos lo anterior entre comillas, expande a "$1 $2 ...", es decir, una sóla cadena con todos los argumentos. La variable $@ expande a los parámetros posicionales (igual que $* pero separados). Sin embargo, entre comillas, "$@", expande a los parámetros posicionales escapados: "$1" "$2"...

Para ver cómo funciona la expansión de parámetros, podemos utilizar el siguiente script (el `for` itera por todos los elementos de la lista, véase las sección 2.13 para más detalles):

Script 2.9: params

```
1  #!/bin/sh
2
3  echo num of args $#
4
5  for i in $*; do
6      echo \* no \" $i
7  done
8
9  for i in $@; do
10     echo \@ no \" $i
11 done
12
13 for i in "$*"; do
14     echo \* \" $i
15 done
16
17 for i in "$@"; do
18     echo \@ \" $i
19 done
```

Al ejecutarlo, imprime:

```
 $  ./xx.sh a "b c" "e f" g
num of args 4
* no " a
* no " b
* no " c
* no " e
* no " f
* no " g
@ no " a
@ no " b
@ no " c
@ no " e
@ no " f
@ no " g
* " a b c e f g
@ " a
@ " b c
@ " e f
@ " g
 $
```

Para el procesado de argumentos, es muy útil el builtin `shift`, que desplaza los parámetros (p. ej. $4 pasará a ser $3) y actualiza el valor de $#. Así puedo extraer los parámetros optativos para que el flujo de código sea igual estén presentes o no. Un ejemplo de cómo funciona se puede ver a continuación:

Script 2.10: params2

```
1  #!/bin/sh
2
3  echo \$0 es $0
4  echo \$\# es $#
5  echo \$\* es $*
6
7  echo $1 $2 $3 $4
8  echo \$\@ es $@
9
10 shift
11 shift
12 echo $1 $2 $3 $4 $#
13 echo \$\@ ahora es $@
```

```
 $ ./param.sh -a -b -c fich
$0 es ./param.sh
$# es 4
$* es -a -b -c fich
-a -b -c fich
$@ es -a -b -c fich
-c fich 2
$@ ahora es -c fich
 $
```

Un ejemplo más avanzado (se entenderá mejor cuando se haya estudiado `if` en la sección 2.13), sería este script con un parámetro optativo `-d`. El script escribe todos los parámetros menos el de depuración. Si está presente, escribe un mensaje por su salida de error:

Script 2.11: tryshift.sh

```
1  #!/bin/sh
2
3  debug=false
4  if [ $# -ge 1 ] && [ $1 = '-d' ]; then
5      shift
6      debug=true
7  fi
8
9  if [ $debug = true ]; then
10     echo estoy depurando 1>&2
11 fi
12
13 echo "$*"
```

```
 $ ./tryshift.sh

 $ ./tryshift.sh -d
estoy depurando

 $ ./tryshift.sh a b c
a b c
 $ ./tryshift.sh -d a b c
estoy depurando
a b c
 $
```

La idea aquí es la siguiente: se extraen los parámetros optativos de la lista de argumentos (como `-d` en este caso) y así el resto del código no necesita ser condicional.

2.11. Agrupaciones de comandos

Podemos ejecutar una *agrupación* de comandos para que se comporte como un sólo comando, por ejemplo, para las redirecciones. Hay dos maneras:

- Ejecución en una *subshell*:

```
( comando; comando; ... )
```

- Ejecución en la misma shell:

```
{ comando; comando; ... }
```

Un ejemplo de agrupación de comandos en la misma shell (ojo, hay dejar un espacio después de la llave abierta, así como terminar cada comando con un punto y coma o final de línea):

```
 $ { echo uno; echo dos; } | tr o 0
un0
d0s
 $ { echo los ficheros de /sys son; ls /sys; } > ficheros
 $ cat ficheros
los ficheros de /sys son
block
bus
class
dev
devices
firmware
fs
hypervisor
kernel
module
power
 $
```

Ejecutar en una subshell es útil para no cambiar el *entorno* de la shell actual (por ejemplo, con `cd` o `export`). Por ejemplo:

```
 $ pwd; ( cd /etc; ls apt; ); pwd;
/tmp
apt.conf.d  sources.list
preferences.d  sources.list.d  trusted.gpg  trusted.gpg.d
/tmp
 $
```

En este caso, vemos que la shell sigue en el directorio `/tmp` al final, aunque se ha usado `cd` para cambiar de directorio. Lo que ha pasado es que se ha cambiado de directorio en la subshell creada y no ha afectado a la shell que estamos usando en el terminal.

En el siguiente ejemplo, la variable `BLA` no existe al final por la misma razón:

```
 $ echo z$BLA; ( export BLA=bla; echo z$BLA; ); echo z$BLA;
z
zbla
z
 $
```

Sin embargo, si ejecutamos lo mismo agrupando en la misma shell:

```
 $ echo z$BLA; { export BLA=bla; echo z$BLA; }; echo z$BLA;
z
zbla
zbla
 $
```

En este caso, al final sí está definida la variable `BLA`.

2.12. Otras sustituciones

2.12.1. Here documents

Los documentos en línea (*here documents*) están a mitad de camino entre una sustitución y una redirección.

Especifica un trozo de texto lo que va leer un comando por su entrada estándar, El trozo de texto se especifica desde `'≪MARCA'` hasta una línea que tiene `'MARCA'`. Por ejemplo:

```
 $ cat <<BLA
 soy un
 here document
 BLA
soy un
here document
 $
```

Nótese que `cat` no ha recibido argumentos. En realidad, es una redirección a la entrada. Veamos otro ejemplo:

```
 $ echo param <<FICH
 hola
 soy un
 fichero
 FICH
param
 $
```

Como `echo` no hace nada con su entrada estándar, vemos que en este caso el *here document* no tiene ningún efecto. Veamos un último ejemplo:

```
 $ wc -l <<EOF
> uno
> dos
> tres
> EOF
3
 $
```

El *here document* tiene tres líneas y `wc -l` las cuenta correctamente.

2.12.2. Globbing

La shell tiene dos minilenguajes integrados en muchos de sus comandos y en la propia shell: expresiones regulares (*regex(7)*) y *globbing* (*glob(7)*). Ambos lenguajes sirven para describir patrones de texto. El globbing es un lenguaje menos expresivo que las expresiones regulares. Se utiliza, sobre todo, para encajar nombres de ficheros y directorios (aunque no sólo se usa para esto). Las expresiones regulares son un lenguaje más expresivo y están presentes en muchas herramientas de programación, incluyendo los editores de programación, IDEs, la shell, bases de datos, motores de búsqueda y muchísimos comandos.

El lenguaje de encaje de patrones de globbing (también conocido como *wildcards* o comodines) utiliza una serie de caracteres especiales para representar diferentes patrones de caracteres:

- **?** cualquier carácter.
- ***** cualquier secuencia de caracteres.
- **[abc]** cualquiera de los caracteres que están dentro de los corchetes (letra 'a', letra 'b' o letra 'c' en el ejemplo), pero sólo uno de ellos.

  Se pueden usar rangos entre los corchetes. Por ejemplo, **[b-z]** cualquier carácter que se encuentre entre esas dos (de la letra 'b' a la 'z' en el ejemplo), pero sólo uno de ellos. Para incluir el '-' literal tiene que ser el primero o el último.
- **[^a-z]** cualquier carácter que no se encuentre en el conjunto (definido como arriba).

Dentro de la shell, una de las sustituciones que se realiza es la de los patrones de globbing por nombres dentro del sistema de ficheros. Como parte de las sustituciones (aparte de las variables, etc.), el último paso es sustituir todos los patrones de globbing que no estén escapados por nombres de ficheros si encajan. Si no encaja ningún fichero, la expresión de globbing no se sustituye, se deja tal cual (a diferencia de una variable, que si no está definida, se sustituye por la cadena vacía):

```
 $ ls
 $ mkdir aa bb cc dd ee fff
 $ touch a1 b2 c3 11 22 33
 $ echo a*
a1 aa
 $ echo [a-c]*
a1 aa b2 bb c3 cc
 $ echo ??
11 22 33 a1 aa b2 bb c3 cc dd ee
 $ echo 7?
7?
 $ echo [a-c][13]
a1 c3
 $ echo q?
q?
 $ echo ???
fff
 $ bla=hola
 $ echo $bla
hola
 $ echo $ble

 $ z='a*'
 $ echo $z
a1 aa
 $
```

En este ejemplo, creamos varios directorios con `mkdir` y varios ficheros vacíos con `touch` dentro de un directorio inicialmente vacío. Luego jugamos sustituciones de variables y globbing. Por ejemplo, como no hay ningún fichero o directorio que encaje con `7?`, el patrón no se sustituye por nada. Lo mismo pasa con `q?`. Sin embargo, vemos que como no existe la variable `ble`, `$ble` se sustituye por la cadena vacía. Observa que en el último caso, se realizan dos sustituciones consecutivas, primero la variable de shell y luego el globbing antes de ejecutar el `echo`.

Una cosa importante es que, a menos que aparezca en el patrón de forma explícita, no se encajan patrones con ficheros y directorios que empiezan por punto (llamados *ficheros ocultos*):

```
 $ touch a ab ac .bla .otro
 $ ls -a
.  ..  .bla  .otro  a  ab  ac
 $ ls
a  ab  ac
 $ echo *
a ab ac
 $ echo .*
. .. .bla .otro
 $ echo * .*
a ab ac . .. .bla .otro
 $ echo [^.]*
a ab ac
 $ echo .[^.]*
.bla .otro
 $
```

2.13. Control de flujo de ejecución

Las condiciones de las estructuras de control de flujo de la shell dependen del estado de salida de un comando que se ejecuta para evaluar la expresión booleana: éxito es verdadero, fallo es falso.

Por ejemplo, el `if`:

```
if comando
then
       comandos
elif comando
then
       comandos
else
       comandos
fi
```

Como vemos, la condición del `if` es la ejecución de un comando.

A continuación vemos un ejemplo de `if` utilizando `test` y la construcción `&&`. Si el fichero `NOTAS` existe y el script ha recibido `-c` como primer argumento, escribe un error (por su salida de errores) y sale con estado de fallo. Si no, calcula la notas, crea el fichero (si se lo han pedido con la opción) y añade las notas al final. Fíjate en las comillas alrededor del primer argumento. Si no están ahí y no hay primer argumento, el comando `test` fallará por que le falta un argumento. Así recibe el argumento cadena vacía. No es lo mismo.

Script 2.12: notas.sh

```
1  #!/bin/sh
2
3  if [ -e NOTAS ] && [ "$1" = '-c' ]; then
4          echo "usage: NOTAS already exists" 1>&2
5          exit 1
6  fi
7  notas=$(./medianotas.sh)
8  if [ $1 = '-c' ]; then
9      echo '#fichero de notas' > NOTAS
10 fi
11 echo $notas >> NOTAS
12 exit 0
```

Se puede negar la condición del resultado de la ejecución de un comando con la admiración. Ten en cuenta que la admiración es una palabra reservada (*keyword*) de la shell, no es un comando.

```
if ! comando
then
       comandos
fi
```

En un script, el final de línea tiene significado: acaba el comando o le da estructura al código. Todas las estructuras de control se pueden escribir separando sus partes mediante un fin de línea o mediante un punto y coma `';'`, como hemos hecho en el script anterior. Lo primero será preferible (y mantener bien la tabulación y demás) en un script. Lo segundo se hará normalmente por comodidad en el uso interactivo de la shell, como se verá en los ejemplos.

En una shell interactiva podría escribir:

```
 $ if test -d /tmp; then echo existe xx; fi
existe
 $ if test -d /xx; then
 echo existe
 fi
 $
```

En un script escribiría mejor:

Script 2.13: ifejem.sh

```
1 #!/bin/sh
2 if test -d /xx
3 then
4     echo existe xx
5 fi
```

O, alternativamente:

Script 2.14: ifejem2.sh

```
1 #!/bin/sh
2 if test -d /xx; then
3     echo existe xx
4 fi
```

La shell también tiene `case`[16]. El `case` puede contener patrones de globbing que se encajarán con la `palabra` indicada. Ten en cuenta que no se encaja con nombres de ficheros, sino con la cadena de texto en la cabecera de la estructura de control:

```
case palabra in
patrón1)
     comandos
     ;;
patrón2 | patrón3)
     comandos
     ;;
*)   # este es el default
     comandos
     ;;
esac
```

Supongamos el siguiente script:

[16]Lo que en otros lenguajes sería la contrucción *switch-case*.

Script 2.15: case.sh

```
#!/bin/sh

palabra="$1"
case $palabra  in
[ab]?)
     echo dos letras empieza por a o b: $palabra
     ;;
?? | ???)
     echo dos o tres letras: $palabra
     ;;
*)
     echo cualquier cosa: $palabra
     ;;
esac
```

Si lo ejecutamos con distintos argumentos:

```
 $ ./case.sh a
cualquier cosa: a
 $ ./case.sh aa
dos letras empieza por a o b: aa
 $ ./case.sh ba
dos letras empieza por a o b: ba
 $ ./case.sh zzz
dos o tres letras: zzz
 $ ./case.sh aaa
dos o tres letras: aaa
 $
```

También tenemos bucles en la shell. La sintaxis del `while` es similar al `if`, itera mientras que el comando ejecutado acabe con estado de fallo:

```
while comando
do
        comandos
done
```

Por ejemplo, esto es un reloj:

```
 $ while sleep 1; do clear; date; done
```

Cada segundo (lo que espera `sleep 1`), el comando `clear` limpia el terminal y después se escribe la fecha actual con el comando `date`. En el terminal, veremos que cambia la

hora cada segundo (como en un reloj). El comando `sleep` siempre sale con éxito (a no ser que se le interrumpa).

El bucle `for` recibe una lista de palabras que asigna a la variable de control en cada iteración:

```
for variable in palabra1 palabra2 palabraN
do
        comandos
done
```

Por ejemplo:

```
 $ for i in uno dos tres; do echo i vale $i; done
i vale uno
i vale dos
i vale tres
 $
```

El comando `seq` nos dará una lista de números si queremos iterar un número concreto de veces. Tenemos que usar una sustitución de ejecución del comando `seq` para que se pueda iterar sobre su salida:

```
 $ for i in $(seq 1 3); do echo i vale $i; done
i vale 1
i vale 2
i vale 3
 $
```

En la shell hay `break` y `continue`. Como siempre, ambos deben usarse con mesura. El código con ellos debe quedar más sencillo que sin ellos, no debe haber una máquina de estados complicada y el nivel de anidamiento debe ser pequeño. Estos principios deben mantenerse cuando se programa en cualquier lenguaje de programación, la shell no es especial en este sentido.

Un apunte muy útil para tareas como renombrar ficheros: el comando `seq` recibe el parámetro `-w` para mantener el ancho de los números constante mediante la adición de ceros a la izquierda:

```
 $ seq -w 8 12
08
09
10
11
12
 $
```

2.14. Comando read: leyendo línea a línea

El comando `read` lee una línea de su entrada estándar y la guarda en la variable que se le pasa como argumento. Se puede utilizar para procesar la entrada línea a línea en un bucle. Sólo debemos hacer eso cuando no tenemos ningún filtro o *pipeline* que nos sirva para hacer lo que queremos.

Por ejemplo, si creamos este fichero:

```
$ echo 'a b
c d' > /tmp/e
$
```

Si ejecutamos el siguiente script, iterará dos veces:

Script 2.16: whileread.sh

```
1 #!/bin/sh
2 while read line
3 do
4   echo $line
5 done < /tmp/e
```

```
$ ./whileread.sh
a b
c d
$
```

Nótese que he redireccionado la entrada de todo el bucle (todos los comandos de dentro del bucle). Si alguno de ellos lee de la entrada, se puede complicar mucho depurar el script. Hay que tener mucho cuidado con estas redirecciones.

En comparación, el siguiente script itera 4 veces, porque corta por espacios y saltos de línea:

Script 2.17: forsinread.sh

```
1 #!/bin/sh
2 for x in `cat /tmp/e`
3 do
4   echo $x
5 done
```

```
$ ./forsinread.sh
a
b
c
d
$
```

En este caso, no estamos procesando línea a línea, sino que se realiza una sustitución y se itera por cada elemento de la salida de `cat`. ¿Qué es un elemento? Para responder a esto, tenemos que ver la variable de entorno `IFS`.

2.15. Variable IFS

La variable *IFS* (***I**nternal **F**ield **S**eparator*) contiene los caracteres que se usan como separadores entre campos. Por omisión contiene el tabulador, espacio y el salto de línea.

Hay que tener cuidado: cambiar el valor de esta variable rompe las cosas, hay comandos que la utilizan. Si vas a modificarla, mejor hacerlo en una subshell para que no afecte al script:

```
 $ for i in $(echo uno dos tres) ; do echo $i ; done
uno
dos
tres
 $ export IFS=-
 $ for i in $(echo uno dos tres) ; do echo $i ; done
uno dos tres
 $ for i in $(echo uno-dos-tres) ; do echo $i ; done
uno
dos
tres
 $
```

2.16. Funciones

En la shell, para estructurar el código, se pueden definir funciones. Se accede a sus parámetros como a los parámetros posicionales de un script. Pero hay que tener cuidado, porque los parámetros locales ocultan los del script (ocultación o *shadowing*).

Por ejemplo:

Script 2.18: shadow.sh

```
1 #!/bin/sh
2
3 hello () {
4         echo hola $1
5         shift
6         echo adios $1
7 }
8 hello uno dos $1
```

Si ejecutamos esto así:

```
 $ ./shadow.sh a b c
hola uno
adios dos
 $
```

Vemos que dentro de la función `hello`, los parámetros posicionales no son los mismos que los del script; son los que se han pasado a la función.

2.17. Alias

Los *alias* son similares a las funciones, pero más sencillos. Están pensados para definir nombres cortos para la shell interactiva, para comandos con argumentos que se ejecutan mucho. Al fin y al cabo, es una sustitución de la shell que definimos nosotros para que sea más sencillo trabajar.

Sin argumentos, el comando `alias` muestra los que hay definidos. El comando `unalias` borra los aliases.

```
 $ alias hmundo='echo hola mundo'
 $ hmundo
hola mundo
 $ unalias hmundo
 $ hmundo
hmundo: command not found
 $  alias
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
 $
```

2.18. Operaciones aritméticas

Para realizar operaciones básicas con enteros podemos utilizar la propia shell, no es necesario usar ningún comando.

Una expresión tras un dólar y con dobles paréntesis, `$((expresión))` se sustituye por la evaluación de la expresión. Por ejemplo:

```
 $ echo $((7 * 8 ))
56
 $ echo $((13 / 2))
6
 $
```

Alternativamente, podemos usar el comando bc[17]. Para realizar operaciones con coma flotante, ya tenemos que recurrir a otros intérpretes como awk (que veremos más adelante) o python.

2.19. Comparación de ficheros

El comando diff compara ficheros de texto línea a línea. Sin embargo, el comando cmp compara ficheros byte a byte, sin suponer que son de texto (es decir, funciona con ficheros binarios).

Ambos comandos tienen éxito si los ficheros son iguales. En caso contrario, salen con estado de salida 1. En caso de un verdadero fallo (argumentos erróneos, ficheros que no existen, etc.), salen con estado 2.

El comando diff escribe por su salida un conjunto de ediciones para convertir un fichero en el otro:

```
 $ echo 'a b
> c d
> e f' > /tmp/x
 $ echo 'c d
> x y
> e f' > /tmp/y
 $ cmp /tmp/x /tmp/y
/tmp/x /tmp/y differ: char 1, line 1
 $ diff /tmp/x /tmp/y
1d0
< a b
3c2,3
< e f
---
> > x y
> > e f
 $ diff /tmp/y /tmp/x
0a1
> a b
2,3c3
< > x y
< > e f
---
> e f
 $
```

[17] Ojo con usar bc para operaciones con decimales. La variable `scale` no hace que se redondee, se trunca y el comando trabaja con precisión arbitraria, no coma flotante.

Las ediciones que propone `diff` se pueden aplicar mediante el comando `patch`, para convertir un fichero en el otro[18]. Además, `diff` se puede aplicar de forma recursiva para comparar dos árboles de ficheros con el parámetro `-r`.

2.20. Filtros y expresiones regulares

2.20.1. Filtros útiles sencillos

El comando `tr` traduce caracteres. El primer argumento es el conjunto de caracteres a traducir. El segundo es el conjunto al que se traducen. El enésimo carácter del primer conjunto se traduce por el enésimo carácter del segundo. Si recibe el parámetro `-d`, sólo recibe un conjunto de caracteres y los borra.

```
 $ cat /tmp/x
a b
c d
e f
 $ tr a-c A-C < /tmp/x
A B
C d
e f
 $ cat /tmp/x | tr -d a-c

 d
e f
 $
```

El comando `sort` ordena las líneas de varias formas. El comando `uniq` elimina líneas contiguas repetidas (tambien puede contarlas con `-c`). El comando `tail` muestra las últimas líneas.

[18] Esta idea es la base del popular sistema de control de versiones `git`.

```
 $ ps
  PID TTY          TIME CMD
25388 pts/9    00:00:00 bash
29301 pts/9    00:00:00 dash
31280 pts/9    00:00:00 ps
31281 pts/9    00:00:00 tail
 $ ps | tail -2        #las dos últimas
31266 pts/9    00:00:00 ps
31267 pts/9    00:00:00 tail
 $ echo 'a b
 c d
 e f
 g h' | tail +3        # a partir de la tercera
e f
g h
```

El comando `tail` tiene también la utilidad de ver qué se va añadiendo a un fichero, por ejemplo de log, mientras sucede con el parámetro `-f` (de *follow*). Con ese parámetro, `tail` se queda esperando, haciendo *polling*[19] del fichero (por defecto cada segundo) y va mostrando lo que se añade al final según sucede:

```
 $ touch fich
 $ cat fich
 $ (sleep 5; date >> fich; sleep 10; date >> fich) &
[1] 7385
 $ tail -f fich
```

Pasados 5 segundos aparecerá una fecha en la salida de dicho `tail -f fich`:

```
vie abr 17 20:33:57 CEST 2020
```

Pasados 10 segundos, la salida ese `tail -f fich` será:

```
vie abr 17 20:33:57 CEST 2020
vie abr 17 20:34:07 CEST 2020
```

Veamos un ejemplo de `sort`. El modificador `-n` indica que la ordenación debe ser numérica (en otro caso, es alfanumérica):

[19] Hacer *polling* es comprobar repetidamente una condición para ver cuando cambia.

```
 $ seq 6 10 | sort
10
6
7
8
9
 $ seq 6 10 | sort -n
6
7
8
9
10
 $
```

El comando `sort` puede recibir el separador de campos cuando se usa el modificador `-t`. También puede recibir una lista de columnas (empezando por la 1) con el modificador `-k`. Se indica la columna de inicio y, opcionalmente, la de final usando una coma después (la clave es un intervalo de campos).

Por ejemplo, en el primer `sort` ordenamos sólo por el campo 2, y en el segundo por los campos 1 al 2. En ambos casos, usamos como separador de campos el guión (que escapamos con una barra invertida):

```
 $ cat x.txt
1-2-4
2-3-3
2-2-1
2-1-4
 $ sort -k2,2 -t\- x.txt
2-1-4
2-2-1
1-2-4
2-3-3
 $  sort -k1,2 -t\- x.txt
1-2-4
2-1-4
2-2-1
2-3-3
 $
```

Sin el parámetro `-s`, la ordenación que realiza no es *estable*. Una ordenación no estable puede cambiar de orden campos que tengan claves iguales. Esto es importante si se realizan varias ordenaciones consecutivas con varios campos diferentes. Si la segunda (y subsiguientes) ordenaciones no son estables, estropearán el orden conseguido anteriormente.

2.20.2. Expresiones regulares

Un lenguaje formal es un conjunto de cadenas de caracteres (puede ser infinito). Hay una familia de lenguajes formales que recibe el nombre de lenguajes regulares. Esos lenguajes se pueden definir mediante expresiones regulares[20].

Se dice que una cadena de texto encaja con la expresión regular si pertenece al lenguaje que describe. Por ejemplo, 'aab' pertenece al lenguaje correspondiente $L = \hat{}a*b\$$, que es L ={'b', 'ab' , 'aab', 'aaab', 'aaaab',...}.

Mediante una expresión regular[21], se define un lenguaje regular para describir y buscar cadenas de caracteres. Son parecidas a los patrones de de globbing, pero con mayor capacidad expresiva. Veremos un dialecto que se llama *extended regular expressions*. Es un estándar de POSIX. Están descritas en `regex(7)`.

Las expresiones regulares no sólo están en la shell. Hay muchos motores de expresiones regulares y dialectos (Perl, Python, etc.). Se utilizan en IDEs y editores, en motores de búsqueda y en muchas aplicaciones. Un buen libro para aprender expresiones regulares es de Fitzgerald [13]. Otro buen libro, más avanzado, es el de Friedl [14].

Como en el globbing, la mayoría de los caracteres se representan a sí mismos salvo un conjunto que tienen significado especial. Una cadena de texto sin caracteres especiales encaja consigo misma, por ejemplo `'abaa'` encaja con la expresión *abaa*. La cadena 'rr`aab`' encaja con la expresión regular *aab* porque `'aab'` con algún prefijo y sufijo es parte del lenguaje.

Los caracteres especiales en las expresiones regulares son:

- Los paréntesis sirven para agrupar (indicar la precedencia de operadores): (e_1).
- El operador alternativa indica alternativas como indica su nombre: $e_1|e_2$.

  Por ejemplo, para dos expresiones e_1 y e_2, $e_1|e_2$ significa que la expresión puede encajar con la primera (e_1) o con la segundo (e_2).
- El operador escape \ sirve para que un carácter pierda su significado especial (esto es, *escaparlo* dentro de la expresión regular).

  Por ejemplo, \(es el carácter paréntesis, aquí no indica una agrupación porque está *escapado*.
- El operador concatenación no se escribe, se indica yuxtaponiendo las expresiones.

  Por ejemplo, una regexp e_1 concatenada a otra regexp e_2, e_1e_2, encaja con una string si una parte p_1 de la string encaja con e_1 y otra parte contigua, p_2, encaja con e_2.

[20] Los lenguajes regulares pueden describirse mediante expresiones regulares y reconocerse de forma rápida y eficiente mediante autómatas. La teoría de lenguajes formales es la que permite tener un motor de expresiones regulares rápido, lo que se consigue construyendo un autómata finito no determinista para reconocer la expresión regular. Más detalles en https://swtch.com/~rsc/regexp/ y [12]

[21] Técnicamente las expresiones regulares son un lenguaje para definir lenguajes, es decir, un metalenguaje.

- Los operadores de conjunto:
  - cualquier carácter, el punto: .
  - conjunto, enumeración de rangos entre corchetes: $[a-c]$
  - conjunto, enumeración de caracteres sueltos: $[acd]$
  - negación de conjunto: $[\hat{}a-c]$

  Por ejemplo `[abc]` encaja con `'a'`. El rango `[a-zA-Z]` que encaja con una única letra, mayúscula o minúscula. El conjunto `[^abc]` *no* encaja con un carácter que sea `'a'` o `'b'` o `'c'`, sin embargo sí encaja con `'z'`.

- Los delimitadores: ^ y $ (principio y fin).

  Por ejemplo `^abc` encaja con `'abcde'` pero no con `'cdabc'`. Como hemos visto antes, sin delimitadores la expresión regular basta que encaje con una subcadena del lenguaje.

- Los operadores de repetición de Kleene $*$, $+$ y el opcional $?$. La expresión e_1* encaja si aparece *cero o más veces* la expresión e_1. La expresión e_1+ encaja si aparece *una o más veces* la expresión e_1. La expresión $e_1?$ encaja con la expresión e_1 o la cadena vacía.

  Por ejemplo, `'aaa'` encaja con la regexp `a*`, pero también con `b*` (por la cadena vacía). También, la cadena `'baaa'` encaja con la regexp `ba+`, la cadena `'bb'` encaja con la regexp `ba*` y finalmente, la cadena `'bb'` no encaja con la regexp `ba+`.

Aunque algunos operadores y construcciones son redundantes, existen por comodidad:

- El conjunto $[a-c]$ es equivalente a $a|b|c$.
- El conjunto $x*$ es equivalente $x+?$.

La interpretación estándar en Unix de encaje de expresiones regulares es avariciosa (*greedy*): encajan con la cadena más larga posible. Es decir, si intento encajar $a*$ (que es la cadena que incluye la cadena vacía, o una cadena con un número indefinido de letras a) sobre la cadena `'aaab'`, aunque técnicamente podría encajar con la cadena vacía o con `'aaab'` o con `'aaab'` o con `'aaab'` o con `'aaab'`, al final va a encajar siempre con la subcadena más larga, es decir, `'aaab'`. De la misma forma $pelota|pel$ encajará en la cadena `'pelota'` con `'pelota'` en lugar de con `'pelota'`[22]

Para limitar qué parte de la subcadena encaja, se pueden utilizar delimitadores como ^ que dicen qué encaja con el principio de la cadena (en un comando orientado a líneas,

[22] Ojo, varias subcadenas separadas pueden encajar, por ejemplo en la cadena `'hola felipe'`, para la expresión regular $hola|felipe$, encajan dos subcadenas, `'hola felipe'` y `'hola felipe'`. En algunos comandos como `sed` se pueden elegir si queremos que encajen la primera o todas. Esto no tiene que ver con el hecho de que si hay una cadena más larga que tiene a la que ha encajado como prefijo que forme parte del lenguaje descrito por la expresion regular, se elija esa (*greedy*) como hemos visto arriba.

el principio de la línea) o $ que encaja con el fin de la cadena. También se pueden utilizar como delimitadores los caracteres que hay antes o después de la cadena que se está intentando encajar.

Es muy importante que siempre busquemos una expresión regular que sea precisa, es decir, que encaje exactamente con las cadenas que buscamos y no con otras.

Veamos dos ejemplos:

- El lenguaje (finito) definido por

$$L = \text{^(hola|adios)[12a-c]\$}$$

  es

$$L = \{\text{'hola1', 'hola2', 'holaa', 'holab', 'holac', 'adios1', 'adios2', 'adiosa', 'adiosb', 'adiosc'}\}$$

  La cadena 'holaantes' no encajaría con la ésta expresión regular, ya que lo impide el delimitador.

- El lenguaje (infinito) definido por

$$L = \text{bueno((malo)*)\$}$$

  define el lenguaje

$$L = \{.*\text{'bueno', } .*\text{'buenomalo', } .*\text{'buenomalomalo'} \ldots\}.$$

  La cadena 'patatabuenomalomalo' encaja con esta expresión regular ya que al no haber delimitador al principio de la expresión, encaja con sufijos de la cadena.

2.20.3. Grep

El comando `grep`[23] es uno de los filtros más útiles de Unix. Sirve para buscar líneas de texto que encajen con una expresión regular. Se puede utilizar pasándole uno o más ficheros como argumento para que filtre las líneas de esos ficheros y las escriba por su salida, o sencillamente como un filtro (escribiendo líneas en su entrada estándar para que escriba por su salida estándar las que encajen). Se usa también para ver utilizando su estado de salida si ha encontrado algo en condicionales.

En realidad, hay tres comandos de la familia de `grep`:

- El comando `grep`, que utiliza expresiones regulares antiguas (no extendidas). No lo usaremos.

[23] El nombre `grep` viene de un comando del editor `ed`, que veremos más adelante.

- El comando `fgrep`, que busca texto literal, no expresiones regulares. Se comporta igual que `grep -F`. Se utiliza cuando quiero buscar un texto literal para no escapar los caracteres especiales. Tiene la ventaja adicional de que es más rápido.
- El comando `egrep`, que busca expresiones regulares extendidas (usaremos este en lugar de `grep`. Se comporta igual que `grep -E`.

El comando `grep` y sus derivados son muy potentes y tienen muchos modificadores. Es interesante ver su página de manual en detalle, *grep(1)*. Los modificadores más útiles son:

- `-v`: realiza lo inverso, esto es, imprime las líneas que no encajan.
- `-n`: indica el número de línea.
- `-e`: indica que el siguiente argumento es una expresión.
- `-q`: silencioso, no escribe nada por la salida (cuando solo nos interesa. el estado de salida).
- `-i`: ignora mayúsculas y minúsculas.

Unos usos muy comunes de `grep` son:

- Ver si una cadena está o no está en un fichero (mirando el estado de salida del comando). En ese caso, seguramente no nos interese su salida y tengamos que usar `-q`.
- Ver qué líneas de un fichero encajan con un patrón (lo escribe en su salida estándar), por ejemplo para contarlas o manipularlas.
- Para obtener los nombres de los ficheros y las líneas que encajan con un patrón. El comando sólo escribe el nombre del fichero en su salida si recibe más de un fichero como parámetro. Un truco estándar es pasar `/dev/null` como parámetro adicional si queremos tener el nombre del fichero también en ese caso.

Es conveniente escapar siempre la expresión regular, puesto que muchos de sus caracteres especiales lo son también caracteres usados por el globbing y la shell. Ten en cuenta que la expresión regular la intepreta `grep`, y si hay una sustitución antes (p. ej. porque la shell inteprete parte como una expresión de globbing) no se buscará lo que pretendes buscar. Para no tener que recordar esos caracteres que se pueden confundir, lo mejor es escapar la expresión regular completa, siempre con comillas simples (a no ser que dependamos de variables para construirla,c).

Para aprender a utilizar `egrep` y experimentar con expresiones regulares, es recomendable utilizar `egrep --color=auto`. De esta forma se pueden ver qué trozos de qué subcadenas encajan con qué cadenas. Además, en `/usr/share/dict` suele haber diccionarios con miles de palabras (una por línea) para hacer pruebas.

Veamos unos ejemplos:

```
 $ alias egcl='egrep --color=auto'
 $ cd /usr/share/dict
 $ egcl 'o.*beca' words
Corabeca
Corabecan
 $ egcl '[eio]n[^aei]cc' < words
inoccupation
nonoccidental
nonocculting
nonoccupant
nonoccupation
nonoccupational
nonoccurrence
 $ egrep -n 'o.*beca' words /dev/null
words:42761:Corabeca
words:42762:Corabecan
 $  egrep -n '[eio]n[^aei]cc' spanish british-english
spanish:37131:equinoccial
spanish:37132:equinoccio
british-english:13981:Pinocchio
british-english:13982:Pinocchio's
 $  ps aux | egrep 'bash$'
paurea   2109  0.0  0.0  19724  5356 pts/0   Ss+  mar25  0:00 bash
paurea   6251  0.0  0.0  19548  5328 pts/2   Ss+  mar30  0:00 /bin/bash
paurea   6287  0.0  0.0  19548  5368 pts/3   Ss+  mar30  0:00 /bin/bash
 $
```

Dos modificadores que en ocasiones resultan muy útiles son `-B` (*before*, antes) y `-A` (*after*, después). Sirven para indicar que se escriban las N líneas de antes o después de la línea que encaja con la expresión. Por ejemplo:

```
 $ seq 1 20 | egrep -B4 '^15$'
11
12
13
14
15
 $ seq 1 20 | egrep -A4 '^15$'
15
16
17
18
19
 $
```

Un último modificador de `grep` se utiliza mucho es `-R` (o, alternativamente, `-r` que no dereferencia enlaces simbólicos[24]) para buscar en todos los ficheros dentro del directorio que se le pase como parámetro y sus subdirectorios. No lo hemos mencionado hasta ahora porque no encaja en el uso de `grep` como filtro, que es la forma en la que es más conveniente pensar en este comando a la hora de programar. El uso como filtro permite combinarlo mediante pipes con otros comandos especializados. En cualquier caso, la búsqueda recursiva puede resultar útil en algún script y se utiliza mucho de forma interactiva. El comando `grep` tiene más modificadores interesantes (por ejemplo `-a` para trabajar con binarios), pero suelen ser más especializados.

2.20.4. Sed

El comando `sed` (***S**tream **Ed**itor*) es un editor de flujos de texto. Está basado en `ed`, un editor de texto basado en comandos que es el tatarabuelo de `vi`. La diferencia con `ed` es que `sed` está diseñado para ser un filtro (de ahí lo de editor de flujos de texto). Es un editor, porque aplica el comando que le damos (`sed` tiene sus propios comandos) a cada línea que lee y escribe el resultado por su salida automáticamente. El modificador `-n` desactiva la impresión automática: sólo se imprimirá lo que indiquemos explícitamente con un comando de `sed`. Si queremos utilizar expresiones regulares extendidas, hay que utilizar la opción `-E`.

Es un editor completo, que permite realizar muchas más cosas que las que vemos aquí. Para más detalles, la página de manual *sed(1)* y el libro de Robbins y Dougherty [15]. Algunos comandos útiles de `sed` son:

- `q`: sale del programa.
- `d`: borra la línea.
- `p`: imprime la línea (es importante en ese caso pasarle también `-n`).
- `r`: lee e inserta un fichero.
- `s`: sustituye cadenas, es el que más se utiliza.

Por ejemplo:

```
$ echo ' a b
c d
e f' | sed 2d
a b
e f
$
```

[24] Explicados en la sección 4.3.3

Aquí, `2` es una dirección. Una dirección es una forma que tiene `sed` de referirse a trozos de la entrada. Lo que hemos ordenado con el comando `2d` es que queremos borrar la línea en la dirección 2 (la segunda línea).

Podemos indicar direcciones así:

- `número` → actúa sobre esa línea.
- `/`e_1`/` → líneas que encajan con la expresión regular e_1.
- `$` → la última línea.
- `número,número` → actúa en ese intervalo.
- `número,$` → desde la línea `número` hasta la última.
- `número,/`e_1`/` → desde la línea `número` hasta la primera que encaje con la expresión regular e_1.

Estos son algunos ejemplos:

- `sed -E '3,6d'` borra las líneas de la 3 a la 6
- `sed -E -n '/BEGIN|begin/,/END|end/p'` imprime las líneas entre esas regexp
- `sed -E '3q'` imprime las 3 primeras líneas.
- `sed -E -n '13,$p'` imprime desde la línea 13 hasta la última.
- `sed -E '/[Hh]ola/d'` borra las líneas que contienen 'Hola' u 'hola'.

El comando sustitución, `sed -E 's/`e_1`/sustitución/'` sustituye **la primera** subcadena que encaja con la expresión regular e_1 por la cadena *sustitución*:

```
 $ echo 'hola hola
 hola hola' | sed -E 's/ho../adios/'
adios hola
adios hola
 $
```

`sed -E 's/`e_1`/sustitución/g'` sustituye **todas** las subcadenas de la línea que encajan con la expresión regular e_1 por la cadena *sustitución*:

```
 $ echo 'hola hola
 hola hola' | sed -E 's/ho../adios/g'
adios adios
adios adios
 $
```

El comando `sed -E 's/(e1)e2(e3).../\2sustitución\1 /g'` utiliza las subcadenas que encajaron con las agrupaciones (los paréntesis en orden de apertura) en la cadena de sustitución. Se llaman referencias hacia atrás, o *backreferences*. En el ejemplo, la cadena que encaje con e_2 se sustituirá por *sustitución*, $e1$ a la derecha, en lugar de \1 y e_3 a la izquierda en el lugar de \2. Se puede referenciar toda la cadena que ha encajado con &.

Un ejemplo de esto puede ser:

```
 $ echo 'hola adios' | sed -E 's/(^h).*adi/\1 bla/'
h blaos
 $ echo 'hola pepe, buenas' | sed -E 's/(^hola )([^,]+).*/adios \2/'
adios pepe
 $
```

En el comando de sustitución, el carácter '/' se puede sustituir por cualquier otro (en todas sus apariciones) y funciona igual. Esto es cómodo para no tener que escaparlo en algunas ocasiones (cuando las cadenas que queremos usar en el comando tienen barras):

```
 $ echo 'hola adios
 mas menos' | sed -E 's/o./es/g'
hesa adies
mas menes
 $ echo 'hola adios
 mas menos' | sed -E 's#o.#es#g'
hesa adies
mas menes
 $
```

A `sed` se le pueden pasar varias expresiones a ejecutar con `-e`. Por ejemplo, para hacer dos sustituciones seguidas

```
sed -E -e 's/^patata$//g' -e 's/hola/adios/'
```

que es equivalente a

```
sed -E -e 's/^patata$//g' | sed -e 's/hola/adios/'
```

pero con un sólo proceso.

Mas ejemplos:

- `sed -E '5!/s/ no / si /' fich.txt` sustituir en todas las líneas menos la 5.
- `sed -E 's_(hola[^ ]) \1_\1_g' fich.txt` eliminar la cadena hola seguida de un no espacio repetida (*backreferences* en el lado izquierdo de la sustitución).

- `sed -E sed 's/[a-zA-Z]/x&x/g' fich.txt` poner una x a cada lado de las letras (*backreference*)
- `sed -En '/^start$/,/^end$/ !p' fich.txt` imprimir todas las líneas salvo las que están entre las líneas 'start' y 'end', la admiración invierte el espacio sobre el que se trabaja (en este caso imprime lo que no encaje).
- `sed '5 r dentro.txt' fich.txt` inserta el contenido del fichero `dentro.txt` a partir de la quinta línea de `fich.txt` (e imprime el resultado por la salida).

2.20.5. Awk

El lenguaje `awk` es un lenguaje de programación de scripting muy sencillo diseñado para la edición al vuelo de texto. Su nombre es un acrónimo de los autores, Al **A**ho, Peter **W**einberger y Brian **K**ernighan, tres de los programadores más famosos de Unix.

Se pueden escribir scripts utilizando el comando `awk` como intérprete directamente mediante el mecanismo de *hash bang*, pero su uso más común es pasándole pequeños programas como parámetro, ya sea en la línea de comando o en scripts.

El comando `awk` lee líneas y ejecuta el programa para cada una de ellas. No imprime por omisión las líneas que lee. Es muy potente, aquí veremos su uso más habitual: imprimir o calcular sobre columnas o registros descritos como texto (p. ej. ficheros csv[25] o ficheros con campos separados por espacios y/o tabuladores).

Las dos funciones principales para imprimir son:

- `print` Función que imprime los operandos. Si se separan con comas, inserta un separador (espacio, tabulador, etc.). Al final imprime un salto de línea. Los argumentos no llevan paréntesis.
- `printf()` Función que imprime con formato. Lleva paréntesis y ofrece control sobre el formato, de una forma similar a la función de `libc` para C.

  El primer argumento es una cadena de formato con *verbos* (`%d` para decimal, `%f` para coma flotante, etc.). Por cada verbo tiene que haber otro parámetro extra cuyo contenido se sustituirá por el verbo en el formato que describe.

También tenemos variables especiales que se pueden usar dentro del programa de `awk`:

- `$0`: La línea que está procesando.
- `$1, $2 ...`: El primer, segundo... campo de la línea.
- `NR`: Número del registro (línea) que se está procesando.
- `NF`: Número del campos del registro que se está procesando (número de columnas).
- `var=contenido`. Se pueden declarar variables dentro del programa. Con el modificador -v se pueden pasar variables predeclaradas al programa.

[25] Un fichero csv es un fichero con campos separados por comas, es un formato muy usado en general.

Por ejemplo:

```
 $ seq 1000 > f.txt
 $ ls -l f.txt
-rw-rw-r-- 1 esoriano esoriano 3893 Oct 15 12:56 f.txt
 $ ls -l f.txt | awk '{ printf("Size: %f KBytes\n", $5/1024)}'
Size: 3.801758 KBytes
 $
```

Aquí hemos usado `printf` para imprimir sólo el tamaño del fichero (campo 5 de la entrada, `3893`). Hemos dividido el tamaño que da `ls` entre 1024 para poder expresarlo en KB.

Veamos otro ejemplo. Para imprimir la tercera y segunda columna de un fichero CSV[26] (el parámetro `-F` indica el separador que va a utilizar `awk` para las columnas, por omisión usa espacios y tabuladores):

```
 $ cat a.txt
123,44,223,23,23,final
14,23,hola,55,7,4
123,23,8,23,bla,45
123,23,3,9,23,9
 $ cat a.txt | awk -F, '{printf("%d\t%d\n", $3, $2)}'
223     44
0       23
8       23
3       23
 $
```

Aquí, estamos filtrando para quedarnos únicamente con los campos 3 y 2, tratándolos como enteros y separados con un tabulador. `awk` es ideal para quitar columnas y reordenarlas a nuestro antojo.

Ten en cuenta que `awk` es un lenguaje con tipado débil. Como se puede ver arriba, si se intenta imprimir algo que no es un número con `%d`, no dará un error, sino que imprimirá 0. De forma similar, todas las variables están inicializadas al valor nulo. Esto lo podemos usar a nuestro favor, pero hay que tener cuidado. En el siguiente ejemplo, puedo no inicializar `tot1` y `tot2` y funcionará bien:

[26] Un fichero CSV (**C**omma **S**eparated **V**alues) es un fichero con campos separados por comas, un formato muy común para hojas de cálculo y *datasets*.

```
 $ cat n.txt
23        34
3         5
5         7
 $ awk '{tot1 += $1; tot2+=$2}
END{printf("%d total1: %d total2: %d\n", NR, tot1, tot2)}'< n.txt
3 total1: 31 total2: 46
 $
```

En este ejemplo, se ven varios fragmentos del programa de `awk`, todos entre llaves. Los trozos entre llaves se ejecutan una vez por línea, menos si tienen `BEGIN` o `END` antes de la llave abierta. En esos casos, se ejecutan una única vez, antes o después de procesar toda la entrada.

En general, un programa de `awk` tiene la siguiente pinta:

```
BEGIN{
        ...
}
guarda {
        ...
}
guarda {
        ...
}
...
END{
        ...
}
```

Donde la *guarda* es una condición que dice si la sentencia debe ejecutarse o no al procesar cada línea. Si no hay guarda, el código entre llaves se ejecutará una vez para cada línea. Por ejemplo, si quiero que se escriba la primera columna de las líneas que escribe `ls -l` que encajan con la expresión regular *[Dd]esktop* podría ejecutar:

```
 $ ls -ld Desktop
drwxrwxr-x 2 paurea paurea 4096 oct 20 10:06 Desktop
 $ ls -l | awk '/[Dd]esktop/{ print $1 }'
drwxr-xr-x
 $
```

Si quiero ser más preciso y decir que se ejecute sólo si la novena columna encaja con la expresión regular, puedo escribir:

```
 $ ls -l | awk '$9 ~ /[Dd]esktop/{ print $1 }'
drwxr-xr-x
 $
```

El operador ∼ sirve para encajar expresiones regulares (evalúa a `true` si encaja y `false` en otro caso). Tiene su negación: !∼.

Puedo utilizar otros operadores de comparación:

```
 $ mkdir nnn; cd nnn
 $ touch a b c d e f g h
 $ ls -l | awk 'NR >= 5 && NR <= 10 { print $9 }'
d
e
f
g
h
 $
```

Fíjate en que también podríamos haber escrito esto, ya que el noveno campo es el último campo de la entrada y `NF` tiene el número de campos:

```
 $ ls -l | awk 'NR >= 5 && NR <= 10 { print $NF }'
d
e
f
g
h
 $
```

El siguiente ejemplo utiliza variables definidas dentro del programa de `awk`:

```
 $  ls -l | awk '{ size=$5; offset=100 ; printf("Size: %08d KBytes\n", offset+size)}'
Size: 00000100 KBytes
Size: 00001479 KBytes
Size: 00020643 KBytes
Size: 00004196 KBytes
 $
```

Ahora usaremos variables pasadas como parámentros al programa de `awk`, empleando el modificador `-v`. Ese modificador nos permite definir variables para el programa:

```
 $  ls -l | awk -voffset=100 '{ size=$5; printf("Size: %08d KBytes\n", offset+size)}'
Size: 00000100 KBytes
Size: 00001479 KBytes
Size: 00020643 KBytes
Size: 00004196 KBytes
 $  ls -l | awk -voffset=200 '{ size=$5; printf("Size: %08d KBytes\n", offset+size)}'
Size: 00000200 KBytes
Size: 00001579 KBytes
Size: 00020743 KBytes
Size: 00004296 KBytes
 $ OFFSET=200
 $  ls -l | awk -voffset=$OFFSET '{ size=$5; printf("Size: %08d KBytes\n", offset+size)}'
Size: 00000200 KBytes
Size: 00001579 KBytes
Size: 00020743 KBytes
Size: 00004296 KBytes
 $
```

Dentro de las llaves, se puede escribir código más sofisticado, con estructuras de control como `if`, `for`, etc. en una sintaxis similar a la de C.

La palabra reservada `next` pasa a la siguiente regla, y `getline` consume la siguiente línea de la entrada. Por ejemplo, para quitar comentarios en un script quitando completamente (incluyendo el final de línea) las líneas que son un comentario entero:

```
 $ cat fich.sh
#!/bin/sh

echo hola # comentario
#comentario
echo adios
 $ awk '/^#.*/{next;} /#.*/{gsub("#.*$","", $0)} {print $0}' < fich.sh

echo hola
echo adios
 $
```

La función `sub` hace una sustitución similar a la de `sed` sin `g` al final. La función `gsub` hace una similar a la de `sed` con `g` al final.

En el lenguaje `awk` hay arrays asociativos que son extremadamente potentes[27]. Por ejemplo, para imprimir cuantos procesos tiene ejecutando cada usuario en el sistema:

[27] Si has programado en Python son similares a sus diccionarios. También son similares a los mapas de Go.

```
 $  ps aux |tail +2| awk '{dups[$1]++} END{for (user in dups) {print user,dups[user]}}'
systemd+ 2
whoopsie 1
message+ 1
colord 1
avahi 2
postfix 2
paurea 212
root 191
 $
```

El `tail` es para quitar la cabecera que añade el comando `ps`. Se puede quitar también sólo con `awk`:

```
 $  ps aux awk 'NR==1{next;} {dups[$1]++} END{for (user in dups) {print user,dups[user]}}'
systemd+ 2
whoopsie 1
message+ 1
colord 1
avahi 2
postfix 2
paurea 211
root 191
 $
```

Otro ejemplo, para imprimir las líneas duplicadas de un fichero:

```
 $cat data
hola
adios
adios
hola
una
dos
tres
hola
cuatro
adios
 $  awk '{dups[$1]++} END{for (line in dups) {if(dups[line]>1) {print line}}}' data
hola
adios
 $
```

Por último, `awk` tiene un soporte (limitado) de funciones. Se puede llamar a una función de `awk` pasando un número menor de argumentos que parámetros tiene definida. Los parámetros que no se encuentran atados, se quedan al valor por defecto (cero). Esto se (ab)usa en `awk` para utilizar los parámetros extra como variables locales (no hay sintaxis para esto). Por convenio los parámetros que se van a utilizar como variables locales se separan mediante un tabulador de los que son auténticos parámetros.

Vemos un ejemplo a continuación (hemos usado el hash bang para definir un *script* que es únicamente de `awk`:

Programa 2.19: func.awk

```
#!/usr/bin/awk -f

function repname(name, repetitions,     i) {
    for(i = 0; i < repetitions; i++) {
        printf("%d: %s\n", i, name);
    }
}

{
    repname($1, 3)
}
```

La función `repname` tiene dos parámetros y una variable local (`i`). Al ejecutar el script repite el primer campo de cada línea 3 veces precedido del número de repetición:

```
 $  echo 'a b
c d
e f'|./func.awk
0: a
1: a
2: a
0: c
1: c
2: c
0: e
1: e
2: e
 $
```

El lenguaje `awk` es muy potente, sólo hemos rascado la superficie. El lenguaje tiene multitud de funciones predefinidas, por ejemplo de matemáticas (números pseudoaleatorios, trigonometría, etc.) y cadenas de texto. Te recomendamos que leas detenidamente la página de manual *awk(1)*. Un buen libro para aprender más sobre `awk` es el de Robbins y Dougherty [15].

2.20.6. Visualización de ficheros

Hay dos comandos similares que se usan para visualizar ficheros (o la entrada, ya que pueden usar al final de un pipeline) de forma interactiva `less` y `more`. El primero permite usar los cursores para hacer *scroll*. El segundo permite paginar pulsando el espacio. Respecto de un editor, tienen la ventaja de que garantizan que no van a modificar el fichero, funcionan en modo texto y son muy rápidos. Cual se debe usar es cuestión de gustos.

Para `less`, el uso básico (tiene muchos más comandos):

- `q`: sale del programa.
- `/regexp`: busca la siguiente línea que encaje con la expresión regular.
- `&regexp`: muestra (filtra) sólo las líneas que encajan con la expresión regular. Una expresión vacía quita el filtrado.

2.21. Árboles de ficheros

A veces es necesario recorrer un árbol entero de ficheros ejecutando comandos sobre cada uno de los ficheros contenidos en los directorios y subdirectorios pertenecientes al árbol.

Hay muchos comandos capaces de recorrrer recursivamente un árbol de ficheros (muchos comandos tienen un modificador `-R` o `-r` a tal efecto). Cuando el parámetro no puede ser `-r` porque eso ya significa otra cosa en ese comando (por ejemplo, en `chmod` eso significa *read*), se suele utilizar la letra mayúscula. Hay que ver de todas formas la página de manual concreta del comando.

Sin embargo, hay que destacar tres que son particularmente útiles para esto: `du`, `ls` y `find`.

2.21.1. Du

El más sencillo y portable de esos comandos es `du` (***d**isk **u**sage*), que sirve para escribir el tamaño de un fichero o directorio. Con el modificador `-a` imprime el tamaño (en bloques) de todos los elementos en un subárbol de ficheros:

```
 $  du -a
4       ./old/README
4       ./old/20_memtest86+
4       ./old/20_linux_xen
12      ./old/00_header
36      ./old
12      ./00_header
20      ./10_linux
4       ./README
4       ./41_custom
4       ./40_custom
4       ./30_uefi-firmware
12      ./30_os-prober
8       ./05_debian_theme
104     .
 $
```

Filtrar la segunda columna con `awk` es una forma sencilla y portable de obtener un listado de ficheros y directorios del subárbol:

```
 $ du -a | awk '{print $2}'
./old/README
./old/20_memtest86+
./old/20_linux_xen
./old/00_header
./old
./00_header
./10_linux
./README
./41_custom
./40_custom
./30_uefi-firmware
./30_os-prober
./05_debian_theme
.
 $
```

2.21.2. Ls recursivo

El comando `ls -aR` también lista un árbol. Desafortunadamente, el formato en el que lo hace no suele ser muy útil para un script (puede estar bien para un humano de forma interactiva):

```
 $ ls -aR
.:
.          05_debian_theme  30_uefi-firmware  README
..         10_linux         40_custom         old
00_header  30_os-prober     41_custom

./old:
.  ..  00_header  20_linux_xen  20_memtest86+  README
 $
```

También podemos generar salida larga:

```
 $ la -laR
.:
total 68
drwxr-xr-x    3 root root   220 abr  3 13:05 .
drwxrwxrwt 1165 root   root   23360 abr  3 13:11 ..
-rwxr-xr-x    1 root root 10627 feb  5  2019 00_header
-rwxr-xr-x    1 root root  6258 oct 17  2018 05_debian_theme
-rwxr-xr-x    1 root root 17123 nov  1 20:16 10_linux
-rwxr-xr-x    1 root root 12059 oct 17  2018 30_os-prober
-rwxr-xr-x    1 root root  1418 oct 17  2018 30_uefi-firmware
-rwxr-xr-x    1 root root   214 oct 17  2018 40_custom
-rwxr-xr-x    1 root root   216 oct 17  2018 41_custom
-rw-r--r--    1 root root   483 oct 17  2018 README
drwxrwxr-x    2 root root   120 abr  3 13:05 old

./old:
total 36
drwxrwxr-x 2 root root   120 abr  3 13:05 .
drwxr-xr-x 3 root root   220 abr  3 13:05 ..
-rwxr-xr-x 1 root root 10627 abr  3 13:04 00_header
-rwxr-xr-x 1 root root 1284 abr  3 13:04 20_linux_xen
-rwxr-xr-x 1 root root  192 abr  3 13:04 20_memtest86+
-rw-r--r-- 1 root root   483 abr  3 13:04 README
 $
```

2.21.3. Find

El comando `find` es una navaja suiza para escribir comandos de shell que trabajan sobre árboles de ficheros.

Se puede ejecutar sobre un directorio para listarlo:

```
$ find .
.
./old
./old/README
./old/20_memtest86+
./old/20_linux_xen
./old/00_header
./00_header
./10_linux
./README
./41_custom
./40_custom
./30_uefi-firmware
./30_os-prober
./05_debian_theme
$
```

Si queremos una salida larga en el listado:

```
$ find -ls
1183 0 drwxr-xr-x   3 root   root   220 abr  3 13:05 .
1195 0 drwxrwxr-x   2 root   root   120 abr  3 13:05 ./old
1206 4 -rw-r--r--   1 root   root   483 abr  3 13:04 ./old/README
1201 4 -rwxr-xr-x   1 root   root  192 abr  3 13:04 ./old/20_memtest86+
1200 16 -rwxr-xr-x   1 root   root 1284 abr  3 13:04 ./old/20_linux_xen
1196 12 -rwxr-xr-x   1 root   root 10627 abr  3 13:04 ./old/00_header
1194 12 -rwxr-xr-x   1 root   root 10627 feb  5  2019 ./00_header
1191 20 -rwxr-xr-x   1 root   root 17123 nov  1 20:16 ./10_linux
1190 4 -rw-r--r--   1 root   root   483 oct 17  2018 ./README
1189 4 -rwxr-xr-x   1 root   root   216 oct 17  2018 ./41_custom
1188 4 -rwxr-xr-x   1 root   root   214 oct 17  2018 ./40_custom
1187 4 -rwxr-xr-x   1 root   root  1418 oct 17  2018 ./30_uefi-firmware
1186 12 -rwxr-xr-x   1 root   root 12059 oct 17  2018 ./30_os-prober
1184 8 -rwxr-xr-x   1 root   root  6258 oct 17  2018 ./05_debian_theme
$
```

Sin embargo, este comando puede hacer mucho más. Tiene dentro un motor de expresiones regulares y otro de globbing. Además, permite ejecutar predicados lógicos sobre los ficheros para decidir qué comando o función ejecuta sobre ellos.

Por ejemplo, para imprimir los ficheros que encajan con una expresión de globbing:

```
 $ find . -type f -name '[12]0*' -print
./old/20_memtest86+
./old/20_linux_xen
./10_linux
 $
```

Otro ejemplo de uso de `find` utilizando una expresión regular (ojo, se aplica sobre toda la ruta, no sobre el último componente como pasa con el globbing):

```
 $ find . -regextype egrep -regex '.*/[0-9].[^/]+' -print
./old/20_memtest86+
./old/20_linux_xen
./old/00_header
./00_header
./10_linux
./41_custom
./40_custom
./30_uefi-firmware
./30_os-prober
./05_debian_theme
 $
```

En el ejemplo estamos utilizando `-regextype` para especificar las expresiones regulares extendidas de `egrep`.

Después de un predicado se puede ejecutar un comando sobre cada fichero que encaje, utilizando las llaves para sustituirlas por el nombre del fichero en el comando, y acabándolo en un punto y coma (que hay que escapar de la shell):

```
 $ find . -regextype egrep -regex '.*/[0-9].[^/]+' -exec echo fich es {}
fich es ./old/20_memtest86+
fich es ./old/20_linux_xen
fich es ./old/00_header
fich es ./00_header
fich es ./10_linux
fich es ./41_custom
fich es ./40_custom
fich es ./30_uefi-firmware
fich es ./30_os-prober
fich es ./05_debian_theme
 $
```

El comando `find` es bastante complicado de utilizar, pero es muy potente. Con `find` también se corre el peligro de terminar escribiendo líneas de comando muy difíciles de entender. Como cualquier gran poder, requiere una gran responsabilidad.

Más información sobre estos tres comandos, como siempre, en *du(1)*, *ls(1)* y *find(1)*.

2.22. Crear comandos en un pipeline

Un patrón común de escritura de scripts de shell es utilizar un *pipeline* para crear un conjunto de comandos y finalmente ejecutarlos. ¿Cómo los podemos ejecutar? Pasándoselos a una shell mediante un pipe. Por ejemplo:

```
 $ ls
JA.txt  Pepe.txt  bla.txt  hola.txt
 $ ls *.txt | sed -E 's/((.*)\.txt$)/mv \1 \2.patata/'| \
awk '{print $1, $2, tolower($3)}'
mv JA.txt ja.patata
mv Pepe.txt pepe.patata
mv bla.txt bla.patata
mv hola.txt hola.patata
 $ ls *.txt | sed -E 's/((.*)\.txt$)/mv \1 \2.patata/'| \
awk '{print $1, $2, tolower($3)}'| \
sh
 $ ls
bla.patata  hola.patata  ja.patata  pepe.patata
 $
```

Esto es útil tanto para programar (es fácil ir generando los comandos a base de reescribir texto) como para comprobar que los comandos están bien de forma interactiva antes de ejecutarlos. Observa que se ha escapado el final de línea con el caracter '\'. Esto se puede hacer tanto en scripts como en la línea de comando para escribir algo en varias líneas y que la shell las interprete como una sóla.

Otro patrón común es generar una lista de argumentos, por ejemplo ficheros, y finalmente ejecutar un comando con todos ellos. A veces se hace esto por eficiencia para no ejecutar un comando por cada fichero. Si quiero hacer algo intermedio y más controlado, e ir ejecutando un comando por cada n argumentos, por ejemplo para no llegar al máximo número de argumentos que requiere un comando o para tener más paralelismo, puedo utilizar el comando `xargs`.

Este comando, lee argumentos a la entrada y cuando tiene suficientes, ejecuta el comando que recibe como argumento sobre ellos:

```
 $ echo a b c | xargs ls -l
-rw-rw-r-- 1 esoriano esoriano 2 mar  1 16:03 a
-rw-rw-r-- 1 esoriano esoriano 2 mar  1 16:03 b
-rw-rw-r-- 1 esoriano esoriano 2 mar  1 16:03 c
 $
```

Con `-n` puede controlar cada cuantos parámetros se ejecuta el comando:

```
 $ ls | xargs -n2 echo lo ejecuto
lo ejecuto bla.patata hola.patata
lo ejecuto ja.patata pepe.patata
lo ejecuto z.patata
 $
```

Si quiero que ejecute en paralelo, puedo utilizar `-P` (hay que tener en cuenta que utilizamos la salida de `ls` para ver cuántos comandos se ejecutan, pero `sh` está ignorando los parámetros extra):

```
 $ ls
a  b  c  d  e
 $ ls | xargs -n2 sh -c 'sleep 10; echo -n $$ " "; date'
10977  mar abr  7 12:49:52 CEST 2020
10980  mar abr  7 12:50:02 CEST 2020
10983  mar abr  7 12:50:12 CEST 2020
 $ ls | xargs -n2 -P2 sh -c 'sleep 10; echo -n $$ " "; date'
10989  mar abr  7 12:51:04 CEST 2020
10990  mar abr  7 12:51:04 CEST 2020
10995  mar abr  7 12:51:14 CEST 2020
 $
```

2.23. Comandos varios

2.23.1. File

El comando `file` utiliza un conjunto de heurísticos (el número mágico, un diccionario de codificaciones, valores e identificadores en varios offsets, el nombre del fichero, etc.) para deducir de qué tipo es un fichero. Una de las formas que tiene de describirlo es utilizando la notación del campo `content-type` del estándar MIME [16], que surgió para embeber contenido multimedia en correos electrónicos. Esta es una forma estándar de identificar tipos de ficheros que se utiliza para muchos propósitos en Unix. Por ejemplo, las imágenes se identifican como `image/xxxxx` donde el lado derecho es el tipo de imagen, por ejemplo `image/gif`.

2.23.2. Join

Un comando muy útil es `join(1)`. Hace un *join* relacional sobre dos columnas (tienen que estar ordenadas). Si quieres saber más de esto, este operador es parte del álgebra relacional que se utiliza en las bases de datos relacionales [17]. Lo más sencillo es verlo con un ejemplo:

```
 $  echo '
 a bla
 b ble
 c blo' > a.txt
 $  echo '
 a ta
 b te
 c to' > b.txt
 $  join a.txt b.txt
a bla ta
b ble te
c blo to
 $
```

Como puedes ver arriba, se funden los dos ficheros juntando las líneas que tienen la columna de referencia (la clave) igual. El comando `join` quita las que no están en alguno de los dos. Esto técnicamente se llama *inner join*. Igual que `sort`, puede recibir diferentes columnas sobre las que trabajar.

2.23.3. Cortar y pegar columnas

Hay dos comandos para cortar y pegar columnas, `cut` y `paste`. Aunque pueden resultar útiles, casi todo se puede hacer con `sed` y `awk`, así que te recomendamos que aprendas primeros esos, que son más potentes, antes de empezar a utilizar estos otros. En cualquier caso, tienen su lugar en el kit de herramientas de un programador de shell. Un ejemplo:

```
 $ echo 'uno dos tres
  cuatro cinco seis' | cut -d' ' -f 1,3
uno tres
cuatro seis
 $ ps | sed 3q > a
 $ seq 1 3 > b
 $ paste a b
PID TTY          TIME CMD        1
8462 pts/4    00:00:00 bash        2
11357 pts/4    00:00:00 ps        3
 $  paste b a
1          PID TTY          TIME CMD
2         8462 pts/4    00:00:00 bash
3        11357 pts/4    00:00:00 ps
 $
```

2.24. Empaquetar ficheros

Un problema importante a la hora de distribuir un conjunto de ficheros (ya sea como uno o varios árboles) es juntarlos en un sólo fichero para distribuirlos. Por ejemplo, podemos necesitar hacer algo como eso en la entrega de una práctica.

Esto es lo que hace el comando `tar`. Su nombre, que significa alquitrán en inglés, viene de la idea de dejar todos los ficheros pegados en el alquitrán. En cualquier comando que guarde árboles de ficheros, hay que guardar los datos (contenido de los ficheros y directorios) y los metadatos (nombres y permisos). Este tipo de herramientas se denominan *archivadores*.

Después de meter un árbol de directorios en un fichero, lo normal es comprimirlo para que ocupe menos. Por eso, las herramientas que saben juntar ficheros normalmente también saben comprimir o llamar a otras herramientas que comprimen (como es el caso de `tar`). El comando `tar` se utiliza muchísimo y es importante saber sus modificadores principales:

- `-c`: Para crear un fichero.
- `-x`: Para extraer los árboles de un fichero.
- `-v`: Para escribir lo que estoy haciendo (verbose).
- `-f` Para pasar un fichero como parámetro para crear o del que leer para extraer.
- `-t` Para listar contenidos pero no escribir nada.

Los ficheros creados por `tar` suelen acaban en `.tar`. Si están comprimidos con `gzip`, suelen terminar en `.tar.gz` o `.tgz`. Si están comprimidos con `bzip2`, suelen ser `.tar.bz` o `.tbz`. En cualquier caso, hay que tener en cuenta que, en general, en los sistemas de tipo Unix un fichero puede tener cualquier nombre independientemente de su contenido (los nombres son *convenios* de uso).

Hay que ser cuidadoso y ejecutar tar desde el lugar adecuado para que al extraer no sobrescriba rutas que no queremos. Si no está bien hecho el fichero tar, se pueden quitar trozos de la ruta, por ejemplo con `--strip-components`. Otros modificadores son:

- `-z`: Ejecutar `gzip` o `gunzip` según corresponda.
- `-j`: Ejecutar `bzip` o `bunzip` según corresponda.

Veamos un ejemplo. Vamos a crear un fichero con `tar` y después lo vamos a extraer en otro sitio:

```
 $ find prueba -ls
 1021 0 drwxrwxr-x 5 paurea paurea 120 abr 7 13:26 prueba
 1029 4 -rw-rw-r-- 1 paurea paurea 84 abr 7 13:23 prueba/fich.txt
 1024 0 drwxrwxr-x 4 paurea paurea 80 abr 7 13:22 prueba/c
 1028 0 drwxrwxr-x 2 paurea paurea 40 abr 7 13:22 prueba/c/e
 1027 0 drwxrwxr-x 2 paurea paurea 40 abr 7 13:22 prueba/c/d
 1023 0 drwxrwxr-x 3 paurea paurea 60 abr 7 13:22 prueba/b
 1026 0 drwxrwxr-x 2 paurea paurea 60 abr 7 13:23 prueba/b/z
 1030 4 -rw-rw-r-- 1 paurea paurea 5 abr 7 13:23 prueba/b/z/fich2.txt
 1022 0 drwxrwxr-x 3 paurea paurea 60 abr 7 13:22 prueba/a
 1025 0 drwxrwxr-x 2 paurea paurea 60 abr 7 13:23 prueba/a/y
 1031 4 -rw-rw-r-- 1 paurea paurea 95 abr 7 13:23 prueba/a/y/otro.txt
 $ tar -czvf prueba.tgz prueba
prueba/
prueba/fich.txt
prueba/c/
prueba/c/e/
prueba/c/d/
prueba/b/
prueba/b/z/
prueba/b/z/fich2.txt
prueba/a/
prueba/a/y/
prueba/a/y/otro.txt
 $ ls -l prueba.tgz
-rw-rw-r-- 1 paurea paurea 416 abr  7 13:26 prueba.tgz
```

```
 $ mkdir ex
 $ cd ex
 $ tar -zxvf ../prueba.tgz
prueba/
prueba/fich.txt
prueba/c/
prueba/c/e/
prueba/c/d/
prueba/b/
prueba/b/z/
prueba/b/z/fich2.txt
prueba/a/
prueba/a/y/
prueba/a/y/otro.txt
 $ find . -ls
 1044 0 drwxrwxr-x 3 paurea paurea 60 abr 7 13:27 .
 1045 0 drwxrwxr-x 5 paurea paurea 120 abr 7 13:26 ./prueba
 1053 0 drwxrwxr-x 3 paurea paurea 60 abr 7 13:22 ./prueba/a
 1054 0 drwxrwxr-x 2 paurea paurea 60 abr 7 13:23 ./prueba/a/y
 1055 4 -rw-rw-r-- 1 paurea paurea 95 abr 7 13:23 ./prueba/a/y/otro.txt
 1050 0 drwxrwxr-x 3 paurea paurea 60 abr 7 13:22 ./prueba/b
 1051 0 drwxrwxr-x 2 paurea paurea 60 abr 7 13:23 ./prueba/b/z
 1052 4 -rw-rw-r-- 1 paurea paurea 5 abr 7 13:23 ./prueba/b/z/fich2.txt
 1047 0 drwxrwxr-x 4 paurea paurea 80 abr 7 13:22 ./prueba/c
 1049 0 drwxrwxr-x 2 paurea paurea 40 abr 7 13:22 ./prueba/c/d
 1048 0 drwxrwxr-x 2 paurea paurea 40 abr 7 13:22 ./prueba/c/e
 1046 4 -rw-rw-r-- 1 paurea paurea 84 abr 7 13:23 ./prueba/fich.txt
 $
```

```
 $ tar -cvf prueba.tar prueba
prueba/
prueba/fich.txt
prueba/c/
prueba/c/e/
prueba/c/d/
prueba/b/
prueba/b/z/
prueba/b/z/fich2.txt
prueba/a/
prueba/a/y/
prueba/a/y/otro.txt
 $ mkdir ex
 $ cd ex
 $ tar -xvf ../prueba.tar
prueba/
prueba/fich.txt
prueba/c/
prueba/c/e/
prueba/c/d/
prueba/b/
prueba/b/z/
prueba/b/z/fich2.txt
prueba/a/
prueba/a/y/
prueba/a/y/otro.txt
 $ find . -ls
 1044 0 drwxrwxr-x 3 paurea paurea 60 abr 7 13:27 .
 1045 0 drwxrwxr-x 5 paurea paurea 120 abr 7 13:26 ./prueba
 1053 0 drwxrwxr-x 3 paurea paurea 60 abr 7 13:22 ./prueba/a
 1054 0 drwxrwxr-x 2 paurea paurea 60 abr 7 13:23 ./prueba/a/y
 1055 4 -rw-rw-r-- 1 paurea paurea 95 abr 7 13:23 ./prueba/a/y/otro.txt
 1050 0 drwxrwxr-x 3 paurea paurea 60 abr 7 13:22 ./prueba/b
 1051 0 drwxrwxr-x 2 paurea paurea 60 abr 7 13:23 ./prueba/b/z
 1052 4 -rw-rw-r-- 1 paurea paurea 5 abr 7 13:23 ./prueba/b/z/fich2.txt
 1047 0 drwxrwxr-x 4 paurea paurea 80 abr 7 13:22 ./prueba/c
 1049 0 drwxrwxr-x 2 paurea paurea 40 abr 7 13:22 ./prueba/c/d
 1048 0 drwxrwxr-x 2 paurea paurea 40 abr 7 13:22 ./prueba/c/e
 1046 4 -rw-rw-r-- 1 paurea paurea 84 abr 7 13:23 ./prueba/fich.txt
 $
```

El comando `tar` sabe llamar a otros comandos de compresión ver *tar(1)*. Por ejemplo, la versión instalada en el sistema en el que estamos escribiendo el libro sabe llamar a `bzip2`, `gzip`, `xz`, `lzip`, `lzma`, `lzop`, `compress` y `zstd`. Hay otros programas que realizan

la doble labor de pegar ficheros y comprimir, muchos de los cuales tienen sus propios formatos para ambas cosas, por ejemplo, `zip`, `rar` y `7zip`.

2.25. Limpiar rutas

Como ya vimos en la introducción, las rutas (*paths*) que recibe un comando pueden ser relativos, o tener concatenaciones de `..`, por ejemplo `../../../xxx` o corresponderse a enlaces simbólicos. Para que sea más fácil trabajar con ellos tenemos el comando `realpath` que limpia (o *canoniza*) un *path*. Por ejemplo:

```
 $ pwd
/tmp/uuu
 $ realpath ../uuu/./../a
/tmp/a
 $ ln -s a zzz
 $ realpath zzz
/tmp/uuu/a
 $ realpath ../uuu/zzz
/tmp/uuu/a
 $
```

Este comando puede resultar muy útil para escribir scripts.

2.26. Creación de ficheros y directorios temporales

En algunos scripts es necesario guardar datos intermedios en ficheros y directorios temporales. El comando `mktemp` sirve para evitar colisiones en las creaciones de estos ficheros o directorios. Creándolos mediante el uso de este comando, se evita que varios scripts o varias ejecuciones del mismo script terminen utilizando el mismo fichero o directorio.

El comando devuelve el nombre del elemento creado. El parámetro `-d` hace que se cree un directorio. Si se le pasa un nombre como parámetro (una plantilla, o *template*) con al menos cinco caracteres `X`, se sustituyen los caracteres por valores aleatorios, dejando el resto del nombre igual.

Es importante recordar borrar los datos temporales (ficheros y directorios) al terminar el script.

Por ejemplo:

```
 $ mktemp
/tmp/tmp.p2UKbVCbVk
 $ ls -l /tmp/tmp.p2UKbVCbVk
-rw------- 1 paurea paurea 0 abr 15 18:16 /tmp/tmp.p2UKbVCbVk
 $ rm /tmp/tmp.p2UKbVCbVk
 $ man mktemp
 $ mktemp mytemp.XXXXXX.yyy
mytemp.98N45v.yyy
 $ ls -l mytemp.98N45v.yyy
-rw------- 1 paurea paurea 0 abr 15 18:16 mytemp.98N45v.yyy
 $ mktemp -d
/tmp/tmp.ChWpjuXjuh
 $ ls -ld /tmp/tmp.ChWpjuXjuh
drwx------ 2 paurea paurea 40 abr 15 18:19 /tmp/tmp.ChWpjuXjuh
 $
```

2.27. Inspeccionar contenido binario de ficheros

Cuando los ficheros contienen datos binarios no imprimibles, se pueden utilizar comandos genéricos o específicos para inspeccionarlos desde la shell. Por ejemplo, el comando `od` para visualizar el contenido en hexadecimal y mostrar los caracteres ASCII si se puede:

```
 $
 $ echo 'a b c
> d e f
> g h i' > zzz.txt
 $ od  -c -tx1 zzz.txt
0000000   a       b       c  \n   d       e       f  \n   g       h
         61  20  62  20  63  0a  64  20  65  20  66  0a  67  20  68  20
0000020   i  \n
         69  0a
0000022
 $
```

Otras herramientas con más opciones de representación son los comandos `xxd` y `hexdump`. Estos comandos hacen algo similar, pero mostrando la representación ASCII en el lado derecho del volcado en hexadecimal, y el desplazamiento en el fichero a la izquierda.

```
 $ echo 'a b
d e' > zzz.txt
 $ hexdump -C zzz.txt
00000000  61 20 62 0a 64 20 65 0a |a b.d e.|
00000008
 $ hexdump -c zzz.txt
0000000   a       b  \n   d       e  \n
0000008
 $
```

Otro comando interesante es `strings`, que muestra las cadenas imprimibles que hay en un binario:

```
 $ strings /bin/ls | sed 5q
/lib64/ld-linux-x86-64.so.2
.j<c~
MB#F-
libselinux.so.1
_ITM_deregisterTMCloneTable
 $
```

Finalmente, para visualizar un fichero como una imagen interpretando los bytes como grises, podemos utilizar `rawtopgm` del paquete `netpbm`. Y para convertir entre formatos dispares de imágenes, `convert` del paquete `imagemagick`. El paquete `imagemagick` es extremadamente útil para automatizar tareas relacionadas con convertir ficheros de imágenes.

Por ejemplo, para ver una imagen en grises de los bytes de ls:

```
 $ sudo apt install netpbm imagemagick
...
 $ rawtopgm -bpp 1 142 144 ./ls
 $ rawtopgm -bpp 1 142 144 ./ls > x.pgm
 $ convert x.pgm x.png
 $
```

El comando `file` muestra información útil sobre el contenido de un fichero. Otra herramienta similar es `exiftool`, que extrae información sobre diferentes formatos de ficheros binarios. El comando `binwalk` permite buscar ficheros embebidos en otros ficheros. Para ingeniería inversa más avanzada, hay herramientas como `radare2`. En realidad, este programa es un desensamblador, pero resulta útil para analizar cualquier tipo de fichero binario. El comando es `r2`.

En general, para cada formato de fichero, suele haber herramientas en la línea de comando para visualizar su formato específico, modificarlo, etc. Por ejemplo, hay muchas

herramientas para ficheros de tipo *PDF* (***P**ortable **D**ocument **F**ormat*): `pdftk`, `pdfinfo`, `pdfimages`, etc.

2.28. Edición automática de ficheros in situ

A veces es necesario editar ficheros in situ (por ejemplo, porque son muy grandes y no se pueden mover). El comando `ed`, es un editor programable (del que viene la idea de `sed` y `vi`)[28].

```
 $ cat zz.txt
hola soy pepe
adios soy juan
hola soy alberto
dime que hora es
dime hola
 $ ed - zz.txt <<EOF
,s/pepe/NO/g
,s/alberto/SI/g
w
EOF
 $ cat zz.txt
hola soy NO
adios soy juan
hola soy SI
dime que hora es
dime hola
 $
```

El comando `ed` recibe comandos propios por su entrada estándar. Mediante la dirección, seleccionamos todo el fichero (como en `sed`, podemos utilizar el mismo tipo de direcciones) y luego sustituimos la expresión regular, con un comando $s/e_1/$sustitución$/g$ equivalente al de sustitución de `sed`. Finalmente, salvamos los cambios de la edición con `w`.

Veamos otro ejemplo:

[28] El comando `ed` es descendiente de `qed`. Ambos editores fueron escritos por Ken Thompson. Ambos eran editores que trabajaban con líneas cuando todavía no había pantallas y se trabajaba con tiras de papel (directamente se iban imprimiendo las líneas de una en una mientras se editaban). El editor `ed` es el primero que introdujo expresiones regulares. El comando `grep` es una simplificación del comando `g/re/p` de `ed` para que cupiese en memoria ejecutar sólo ese comando, buscar una expresión regular en el fichero cuando el fichero y el programa no cabía entero en memoria (en esa época, la memoria era del orden de KB).

```
 $ cat otro.txt
hola soy pepe
BEGIN
adios soy juan
hola soy alberto
dime que hora es
END
dime hola

 $ echo '
/BEGIN/, /END/d
w
'|ed -- zz.txt
84
?
25

 $ cat otro.txt
hola soy pepe
dime hola

 $
```

Finalmente, veamos un ejemplo con lo que fue la génesis de `grep`[29]:

```
 $ cd /usr/share/dict
 $ echo 'g/^x...hos/p' | ed -s words
pyoxanthose
xanthosiderite
xanthosis
xanthospermous
 $
```

Otra forma de editar ficheros *in situ* es utilizar `sed -i` (ojo, no todo se puede hacer in situ, hay que saber bien lo que se está haciendo). Por ejemplo:

[29] El nombre de `grep` viene de ese comando de `ed`, g, expresión regular, p.

```
 $ cat x.txt
a b c patata c d e
patata a b c
c d e patata
 $ sed -E -i "s,patata(.*),\1patata,g" x.txt
 $ cat x.txt
a b c  c d epatata
 a b cpatata
c d e patata
```

2.29. La shell para automatizar tareas

Como hemos dicho al principio de este capítulo, la shell es una herramienta fundamental para automatizar tareas. En esta sección veremos usos más avanzados de la shell para ver el tipo de cosas que se pueden llegar a hacer, y sobre todo, cuando *no se debe utilizar la shell*.

2.29.1. Seguridad: cuando no debes usar la shell

Las mismas propiedades que hacen de la shell una herramienta potente (utilización flexible de texto, facilidad de escribir metacódigo, esto es, líneas de shell generadas al vuelo, multitud de pequeñas herramientas interconectadas, etc.) hacen que utilizar programas en los que el programa que requieren un de privilegio alto (esto es, permisos para hacer cosas peligrosas, cuenta de `root`, etc.) sea muy peligroso.

Hay que ser muy cuidadoso con los scripts de shell que ejecutan en servidores o como usuarios con alto nivel de privilegio. La shell es extremadamente útil, pero tiene sus limitaciones. Aunque se sea cuidadoso con la *higienización* de la entrada del usuario, es fácil cometer errores y sufrir algún tipo de *ataque de inyección* de shell en el que el usuario ejecuta código no deseado.

2.29.2. Automatización de tareas

La filosofía de Unix, de la cual la shell es el epítome, es que muchas herramientas interconectadas que se comunican mediante un formato común de texto permiten automatizar tareas y escribir pequeños programas extremadamente potentes. En esta sección vamos a ilustrar estas ideas con algunos programas y estrategias que se utilizan de forma habitual para ahorrar tiempo y trabajar en el día a día.

Make

El programa `make` surgió como una herramienta para la recompilación en demanda para el lenguaje de programación *C*, pero es mucho más general. Podemos utilizar `make` siempre que queramos tener una receta (de shell) para crear un fichero. El comando

hace uso de un fichero con la descripción de recetas `Makefile`. El comando `make` puede ejecutar sin argumentos o con una lista de objetivos a crear y busca un fichero `Makefile` de recetas con ese nombre en el directorio actual.

Un ejemplo de receta sería:

```
1 tarta.txt: huevos.txt leche.txt harina.txt
2     batir huevos.txt > batidos.out
3     mezclar batidos.out leche.txt  harina.txt -f salida.out
4     hornear salida.out > tarta.txt
5     rm *.out
```

La línea 1 es la que dice el fichero que se va a generar `tarta.txt` y los ficheros de los que depende son, en este ejemplo, `huevos.txt` `leche.txt` y `harina.txt`. Si cualquiera de estos ficheros se modifica, es decir, su fecha de modificación es más actual que el fichero `tarta.txt`, cuando ejecutemos `make`, la regla correspondiente se ejecutará. Si los ficheros no existen y hay una regla para generarlos, también se ejecutará dicha regla. Las líneas 2–5 son los comandos que se ejecutarán cuando se dispare esta regla. Sus líneas deben ser contiguas (sin líneas vacías en medio) y precedidas por un tabulador. Lo interesante de este comando es que escribo la regla una única vez y más adelante, cuando necesite que se generen los ficheros, ejecutaré `make` o `make tarta.txt`. En ese momento, se hace lo necesario para *hacer la tarta*, ejecutando sólo las reglas que haga falta.

Un ejemplo más completo y real podría ser este fichero `Makefile`, usado para compilar un fichero en LaTeX[30] con figuras.

```
1  all: libro.pdf
2  # esto es un comentario, y $$ es una forma de escapar $
3  FIGS=$(shell ls figs/*.svg|sed -E 's/\.svg$$/.pdf/g')
4  TEXS=$(wildcard *.tex)
5
6  libro.pdf: libro.tex $(FIGS) $(TEXS) libro.bib
7      pdflatex -interaction=nonstopmode libro.tex
8      bibtex libro
9      pdflatex -interaction=nonstopmode libro.tex
10     pdflatex -interaction=nonstopmode libro.tex
11
12 figs/%.pdf: figs/%.svg
13     inkscape -D $< --export-filename=$@
14
15 .PHONY: clean nuke all
16
17 clean:
18     rm -f libro.nav libro.snm libro.bbl libro.lol
19
20 nuke:
21     rm -f libro.pdf figs/*.pdf
```

[30] LaTeXes un sistema de composición de textos.

Vamos a ir paso a paso explicándolo. La línea 1 es una regla vacía. Si ejecutamos `make` sin argumentos, ejecutará la primera regla que se encuentre. En este caso, es equivalente a ejecutar `make all` que tiene como dependencia el fichero `libro.pdf`. La línea 2 es un comentario, se ponen igual que en un script (la almohadilla comenta toda la línea). La línea 3 define una variable dentro del `Makefile`. El contenido de la variable se obtiene al ejecutar una línea de shell, que en este caso buscará todos los nombres de los ficheros en el directorio `figs` con extensión `svg` sustituyendo su extensión por `pdf` (que es lo que necesita el fichero de LaTeXpara incluirlos como figuras). Estos nombres, los meterá en la variable `FIGS`,. El `$` de la expresión regular está dos veces para escaparlo y que `make` no haga una sustitución. La siguiente línea sencillamente mete en `TEXS` todos los nombres de los ficheros de extensión `tex`. La línea 6 es la regla para generar el fichero `libro.pdf` que depende de todas las figuras, los ficheros `tex` y el fichero `bib`. Si cualquiera de ellos cambia hay que aplicar la regla (líneas 7-10, que ejecuta los comandos `pdflatex` y `bibtex`).

La siguiente regla, en la línea 12, es más interesante. Cuando `make` necesite cualquier fichero que encaje con `figs/*.pdf` (que en el Makefile se escribe con %), éste dependerá del fichero con el mismo nombre pero acabado en `svg`. Lo que ha encajado en el % se sustituye en la izquierda por el % de la derecha. A su vez, en la línea 13, el comando de shell de la regla, recibe como parámetro `$<` que se sustituye antes de ejecutar por el primer prerrequisito (en este caso, `figs/XXX.svg` con XXX lo que sea que haya encajado en la llamada) y `$@` se sustituye por el fichero objetivo a crear (en este caso, `figs/XXX.pdf`). Por ejemplo `make figs/a.pdf` intentará crear ese fichero si `figs/a.svg` es más actual y en ese caso, ejecutará:

```
inkscape -D figs/a.svg --export-filename=figs/a.pdf.
```

La línea 15, que empieza por `.PHONY`, dice qué reglas **no** tienen como objetivo crear un fichero (para que no se mire si es más moderno). Por ejemplo, se usa para las reglas para limpiar (en el ejemplo `clean` en la línea 17 limpia ficheros intermedios y `nuke` en la línea 20 limpia todos los ficheros creados) y la regla `all` de la línea 1 que ya hemos mencionado. `.PHONY` indica que no es necesario crear ningún fichero `clean`, `all` o `nuke`, que son sólo los nombres de la regla.

A continuación podemos ver una sesión de shell utilizando este `Makefile`. Fíjate que sólo se ejecutan las reglas es necesario ejecutar para regenerar los objetivos que hayan cambiado (cuando los ficheros dependencia son más modernos que el objetivo o siempre si la regla es `PHONY`).

```
 $  ls
Makefile  a.tex  b.tex  figs  libro.bib  libro.tex
 $  ls figs
a.svg
 $  make
inkscape -D figs/a.svg --export-filename=figs/a.pdf
pdflatex -interaction=nonstopmode libro.tex
bibtex libro
pdflatex -interaction=nonstopmode libro.tex
pdflatex -interaction=nonstopmode libro.tex
 $  make clean
rm -f libro.nav libro.snm libro.bbl libro.lol
 $  make
make: Nothing to be done for 'all'.
 $  ls
Makefile  a.tex  b.tex  figs  libro.bib  libro.pdf  libro.tex
 $  make nuke
rm -f libro.pdf figs/*.pdf
paurea@myo:/tmp/z$ make
inkscape -D figs/a.svg --export-filename=figs/a.pdf
pdflatex -interaction=nonstopmode libro.tex
bibtex libro
pdflatex -interaction=nonstopmode libro.tex
pdflatex -interaction=nonstopmode libro.tex
paurea@myo:/tmp/z$ make
make: Nothing to be done for 'all'.
paurea@myo:/tmp/z$ touch figs/a.svg
paurea@myo:/tmp/z$ make
inkscape -D figs/a.svg --export-filename=figs/a.pdf
pdflatex -interaction=nonstopmode libro.tex
bibtex libro
pdflatex -interaction=nonstopmode  libro.tex
pdflatex -interaction=nonstopmode libro.tex
paurea@myo:/tmp/z$ touch b.tex
paurea@myo:/tmp/z$ make
pdflatex -interaction=nonstopmode libro.tex
bibtex libro
pdflatex -interaction=nonstopmode  libro.tex
pdflatex -interaction=nonstopmode libro.tex
paurea@myo:/tmp/z$
```

Con esto nos hacemos una idea de la funcionalidad. Hay cosas más avanzadas en el formato `Makefile`, reglas automáticas para compilar ficheros, sustituciones, formas de

llamar a `make` desde una regla etc.

Es una herramienta extremadamente útil siempre que haga falta crear ficheros mediante la shell, en demanda, con un esquema de dependencias (esto es, que no se creen los ficheros si no es necesario, porque no hay cambios). En general, esto es necesario cuando tenemos un programa (p. ej. un programa en C), documento (p. ej. un texto en LaTeX) o proyecto con múltiples ficheros fuente. Muchos lenguajes tienen sus propios comandos que saben las dependencias del lenguaje (por ejemplo Go), pero otros no y a veces hay proyectos multilenguaje (un libro como este escrito en LaTeXy con código C, shell, etc.) o documentos con multiples formatos intermedios, figuras, conversiones de ficheros de log, etc. En estos casos `make` es extremadamente útil.

La página de manual de `make` no es muy detallada, pero la de `info`, un sistema alternativo de documentación, es totalmente detallada. Esto sucede con algunos comandos de GNU. En Ubuntu, si instalas `make-doc` (por ejemplo con `apt install make-doc`), tendrás acceso a esas páginas[31]. La documentación navegable mediante el comando `info` funciona con enlaces parecidos a los de la web. Aunque es un poco rara, es fácil hacerse a ella. Para más detalles, ejecuta `info info` e `info make`.

Herramientas variadas

Muchas tareas de red se pueden automatizar mediante comandos de shell. Algunos comandos interesantes son `netcat`, `whois`, `ping`, `dig` y `arping`. Si queremos interactuar con una máquina a través de la red mediante un protocolo estándar para averiguar información, traducir un nombre, etc. seguramente se puede hacer mediante un comando de shell. En muchos de estos comandos, es importante asegurarse de que se pone un *timeout* razonable. Por ejemplo si quiero ejecutar un comando como `ps` en una máquina remota y detectar si ha habido un error, puedo ejecutar:

```
ssh -t -o ConnectTimeout=1 maq.bla.es ps
```

Así, puedo asegurarme de que, si no puedo contactar con la máquina, obtengo un error en un tiempo razonable y puedo informar al usuario. En caso de fallo en la conexión, `ssh` sale con estado 255.

Si necesito interactuar con una página web (interactuar con un formulario, bajarme el árbol de páginas web de forma recursiva, etc.) puedo usar `wget` o `curl`. Por ejemplo, para traerse una lista de máquinas e la web de laboratorio e imprimirlas:

[31] Suponiendo, claro que tienes instalado `make`, que junto con muchas herramientas básicas de programación, forman parte del paquete `build-essential` en Ubuntu

Programa 2.20: fgreat.sh

```
#!/bin/sh

FNAME=`mktemp mytemp.XXXXXX.yyy`
rm $FNAME 2>/dev/null
cd /tmp
URL=https://labs.etsit.urjc.es/parte/index.php

if ! curl --connect-timeout 3 $URL > $FNAME 2>/dev/null; then
        echo "cannot reach $URL" 1>&2
        rm $FNAME
        exit 1
fi

# File in one line, try to break by machines, match names
# fragile, but we have to be careful to select the right
# machines or the firewall will eat us.
names=`cat $FNAME |tr '\n' ' '\
        |sed -E 's/arriba/arriba\n/g'|sed 's/quina.*SSH//g'\
    |grep arriba\
        |grep -E 'f-l[0-9]+-pc[0-9]+'\
    |sed -E 's/.*(f-l[0-9]+-pc[0-9]+).*/\1/g'|sort -u`
echo $names
rm $FNAME
```

En ese script estamos procesando una página sobre cuyo formato no tenemos ningún control. Esto se llama *web scraping*, y es poco recomendable si queremos tener sistemas robustos. Sin embargo, para automatizar pequeñas tareas sabiendo que habrá que cambiar el script de vez en cuando, puede ser muy potente.

Otro uso interesante es la sincronización de sistemas de ficheros. El comando `rsync`, es una navaja suiza en este campo. Se puede utilizar para hacer backups, copiar ficheros de forma incremental a través de la red, mantener varias *homes* sincronizadas, etc.

El comando `git` es fundamental para el desarrollo de proyectos de programación, especialmente a nivel colaborativo. Este comando está escrito específicamente para ser utilizado en la shell y es especialmente fácil automatizar tareas con él. La página de manual de `git` es muy detallada *git(1)*, pero te recomendamos que leas un libro antes para entender el esquema general. El libro "Pro Git" está una buena referencia y está disponible online de forma gratuita en https://git-scm.com/book/en o se puede obtener en papel [18].

La edición de texto es un campo en el que esta aproximación de utilizar texto plano y herramientas generales es muy potente. En particular, el uso de un lenguaje de marcado como LaTeX (mencionado anteriormente) permite describir lo que se quiere escribir en texto y el formateado lo hace el propio programa (en base a las marcas usadas). Podemos *compilar* LaTeX, para general el documento formateado, mediante los comandos `latex` y `pdflatex` (el segundo es más común en la actualidad). Aunque en un sistema como LaTeXel formato es más rígido y tiene una curva de aprendizaje mayor, las ventajas son fundamentales:

- Decisiones avanzadas de formateado y edición se encuentran ya *cableadas* en el sistema o se pueden importar mediante estilos de forma sencilla.
- Se pueden utilizar herramientas de shell (por ejemplo `grep`) y de desarrollo, (como `git`) para editar tanto a nivel local como colaborativo.
- En general se desacopla la edición de la escritura, perdiendo mucho menos tiempo en el bucle *modifico, resitúo figura a mano, vuelvo a colocar el texto,* etc.

Hemos escrito este libro utilizando LaTeX.

Para hacer diagramas de diferentes tipos en una shell, te recomendamos que mires también el paquete `graphviz`.

También hay herramientas muy útiles para convertir, inspeccionar y juntar ficheros de diferentes formatos, como por ejemplo `pdfjam` (parte del paquete `tex-extra-utils` en Ubuntu). Estos programas permiten automatizar tareas básicas. Por ejemplo, para pegar ficheros en formato PDF para entregarlos a la administración, se puede usar el siguiente alias:

```
 $ alias pdfjamA4="pdfjam --fitpaper true --paper a4paper"
 $ alias pastepdf="pdfjamA4 --rotateoversize false  --suffix joined"
 $  ls
dos.pdf  uno.pdf
 $ pastepdf *.pdf
 $   ls
dos.pdf  uno-joined.pdf uno.pdf
```

El fichero `uno-joined.pdf` contiene los ficheros pdf pegados uno detrás de otros en un pdf único válido.

Otro paquete interesante es `imagemagick`, que contiene el comando `convert`. Ese comando permite convertir entre cientos de miles de formatos de imagen, rescalando, filtrando, cambiando formatos y parámetros, etc. Se pueden modificar las imágenes in situ (cuidado, se pierde la imagen antigua) con `mogrify` (es equivalente a `convert`, pero in situ).

Aparte de todo esto que hemos dicho, muchas de las aplicaciones interactivas de usuario se pueden ejecutar sin interfaz de usuario para automatizar tareas (se le suele llamar ejecución en modo *batch*). Por ejemplo, uno de los programa de edición de gráficos vectoriales más populares de Linux es `Inkscape`. Este programa se puede ejecutar para que convierta ficheros tipo SVG a PDF mediante:

```
inkscape -D dibujo.svg --export-filename=dibujo.pdf
```

Algo similar se puede hacer con `gimp`, un programa de retocado de imágenes, ejecutándolo con `-b`. Ambos programas se pueden automatizar escribiendo en un lenguaje de programación (Python, Lisp) y luego llamando a los comandos en modo batch. Imagina, por ejemplo, que quieres quitar los ojos rojos de 200 imágenes cada día. Puedes hacer un script que llame al procedimiento de quitar ojos rojos de `gimp`. Para que te

puedas hacer una idea (puedes utilizar parte de lo que hay aquí para construir el tuyo propio, pero lo suyo es que mires los tutoriales), los pasos a seguir serían:

1. Mirar la versión de gimp: `gimp -v`. La de ejemplo es la 2.10.
2. Ejecutar gimp y mirar en el *procedure browser* (en el menú ayuda p *help*) qué función hay para eso, que es `plug-in-red-eye-removal`.
3. Hacerse un script que cargue las imágenes, las salve y demás en script-fu (un dialecto de Lisp) o en Python. El ejemplo en Lisp es:

```
(define
    (paurea-redeye pattern threshold)
    (let* (
        (filelist (cadr (file-glob pattern 1)))
        )
    (while (not (null? filelist))
        (let* (
            (filename (car filelist))
                (image (
                        car (gimp-file-load RUN-NONINTERACTIVE filename filename))
                    )
                (drawable (car (gimp-image-get-active-layer image)))
            )
            (plug-in-red-eye-removal RUN-NONINTERACTIVE image drawable threshold)
            (gimp-file-save RUN-NONINTERACTIVE image drawable filename filename)
            (gimp-image-delete image)
        )
        (set! filelist (cdr filelist))
    )
    )
)
```

   Se salva en donde tiene `gimp` su configuración, `$HOME/.config/GIMP/2.10/scripts`.

4. Se ejecuta con:

   ```
   gimp -i -b '(paurea-redeye "*.jpeg" 10)' -b '(gimp-quit 0)'
   ```

   Fíjate que el programa de arriba admite globbing. Esto permite correr únicamente una instancia de gimp (que es bastante pesado). En cualquier caso, si no funcionase, podría poner un bucle en la shell y tardaría un poco más. Por supuesto, la llamada puede formar parte de un script de shell.

5. Se puede depurar todo esto con `gimp -i -b -` de forma interactiva o metiendo en `fich` todo:

   ```
   cp redeye.scm fich
   echo '(paurea-redeye "red.jpeg" 10)' -b '(gimp-quit 0)' >> fich
   ```

   Se puede ejecutar `gimp -i -b - < fich`.

Por último, cuando escribas programas, tanto para tí como para otros, en cualquier lenguaje de programación, acuérdate de estos principios:

- Intenta que utilicen formatos de texto abiertos.
- Que se puedan llamar cómodamente desde la shell.
- Que haya formas de llamarlos en modo batch (sin interfaz de usuario).
- Que sean programables.

Todo esto hará que sean más útiles, más fáciles de probar y en general hará tu tarea como desarrollador, más agradable.

2.30. Ejercicios resueltos

A continuación se muestran varios ejercicios de shell resueltos.

2.30.1. Fgreat

Escribe un script `fgreat.sh` que muestre un comando para renombrar el fichero más grande del directorio actual, añadiendo al final de su nombre `.old`. Con el parámetro `-r` buscará el fichero de forma recursiva.

Si sucede cualquier error o los parámetros son incorrectos, los scripts deben escribir el error por la salida de error y salir con estado de error.

Un ejemplo de ejecución podría ser:

```
 $ pwd
/tmp/z
 $ ls -l
total 12
-rw-rw-r-- 1 paurea paurea 5 abr  8 13:53 gra.txt
-rw-rw-r-- 1 paurea paurea 4 abr  8 13:53 med.txt
-rw-rw-r-- 1 paurea paurea 2 abr  8 13:53 peq.txt
 $ fgreat.sh
mv /tmp/z/gra.txt /tmp/gra.txt.old
 $ pwd
/tmp/z
 $  fgreat.sh -r
mv /tmp/z/x/gig.txt /tmp/z/x/gig.txt.old
```

Una solución a este ejercicio, se puede ver a continuación:

Programa 2.21: fgreat.sh

```
1  #!/bin/sh
2
3  usage(){
4      echo "usage: $0 [-r]" 1>&2
5      exit 1
6  }
7  nofiles() {
8      echo error: no files 1>&2
9      exit 1
10
11 }
12
13 listfiles(){
14     ls -la |egrep '^-'|awk '{print $5, $9}'
15 }
16
17 cmd=listfiles
18 if [ $# = 1 ] && [ $1 = '-r' ]; then
19     shift
20     cmd='du -ba .'
21 fi
22
23 if [ $# != 0 ]; then
24     usage
25 fi
26
27 if [ $(ls -la|wc -l) -le 3 ]; then
28     nofiles
29 fi
30
31 files=$($cmd |sort -nr|awk '{print $2}')
32 biggest=''
33 for i in $files;
34 do
35     if [ -f "$i" ]; then
36         biggest=$(realpath "$i")
37         break
38     fi
39 done
40
41 if [ -f "$biggest" ]; then
42     echo $biggest| sed -E 's/(^[^ \t]+)/mv \1 \1.old/g'
43 else
44     nofiles
45 fi
```

El programa tiene dos partes claramente diferenciadas, una en la que se comprueban los argumentos (hasta la línea 29, incluida) y una en la que se hace el trabajo, (de la línea 30 hasta el final).

Procesado de argumentos

Primero se definen dos funciones, para dar un mensaje de error y salir. La primera, `usage`, sirve para el caso de que hayan llamado mal al comando. Esta función, o una similar, estará en muchos de los scripts que escribamos. La segunda se utiliza cuando no hay ficheros en el directorio actual. Ambas escriben un mensaje por la salida de error y salen con estado de error.

El primer paso es comprobar si el script ha recibido el único argumento posible, `-r`, que es opcional. La idea es que, en ese caso, se prepare todo para que el resto del script sea el mismo, dejando en algunas variables lo necesario para proceder. En el script que nos ocupa, lo que vamos a hacer es obtener un listado de ficheros y su tamaño, ordenarlos por el valor numérico del tamaño, quedarnos con el más grande y generar el comando. La única parte diferente entre el comportamiento recursivo y el no recursivo (con y sin el modificador `-r`) será la de conseguir el listado de ficheros sólo de un directorio o de un directorio y sus subdirectorios.

En caso de no ser recursivo, ejecuta la función `listfiles`. Esta función lista todos los ficheros y directorios del directorio actual, filtra sólo los ficheros y se queda con las columnas correspondientes al tamaño y al nombre. Estamos suponiendo que no hay espacios ni tabuladores en el nombre. Aunque esto se podría tratar, complica innecesariamente el script. Supondremos que los nombres son razonables esto salvo que sea trivial comprobarlo. En general, introducir espacios de diferentes tipos (espacios, tabuladores, fines de línea) en los nombres de ficheros complica innecesariamente todo el uso de la shell. En la práctica, antes de ejecutar un script así, habría que renombrar ficheros y directorios para que tengan nombres razonables o complicar mucho el script para que los considerase. La función `listfiles` se podría haber implementado alternativamente listando únicamente los ficheros mediante `find` así: `du -b $(find . -maxdepth 1 -type f)` o así: `find . -maxdepth 1 -type f -exec du -b \{\} \;`.

En el caso de que el enunciado especifique que el script actúe de forma recursiva, es decir, sobre todo el árbol, habría que ejecutar `du -ba .` sobre el directorio actual `'.'`. El modificador `-b` sirve para para obtener la cuenta en bytes y `-a` para que el comando actúe de forma recursiva.

El comando a ejecutar se guarda en una variable (línea 17). Si hay un argumento y es `-r` (línea 18), se extrae de la lista de argumentos mediante `shift` (línea 19) y se cambia la variable que contiene el comando (línea 20). A partir de ahora el resto del script es común para ambos casos.

La estrategia de procesar argumentos por separado es muy importante por dos razones. En primer lugar, permite comprobar bien los argumentos. Cualquier entrada proveniente del usuario debe ser comprobada en detalle por cualquier programa, por si fuese errónea. En segundo lugar, simplifica el resto del programa, que tiene ya su comportamiento segregado en un conjunto de casos, con la parte común en un sólo sitio. Si se mezcla el procesado de argumentos y la lógica del programa, es muy fácil acabar con un árbol de condiciones complicado y lógica repetida.

Como ya hemos extraído el argumento opcional (en caso de existir) y el script no recibe más argumentos, si ha recibido alguno, acaba con error (líneas 21-23).

Por último, una vez comprobada la corrección formal de los argumentos (que como mucho se ha recibido un argumento y es el que tiene que ser), se comprueba el argumento implícito: el directorio actual. Si el directorio actual no tiene ficheros o directorios (el resultado de `ls -la` es de tres líneas la cabecera, `'.'` y `'..'`) el script acaba con error (línea 28).

Lógica del script

Tras procesar los argumentos, se ejecuta la lógica del propio script. El primer paso es ordenar numéricamente al revés por la primera columna (el tamaño) dejar sólo la segunda columna (el nombre) y guardar el resultado en una variable (línea 31). A continuación, se recorre la lista de nombres hasta encontrar un fichero. Esto es importante hacerlo porque `du -ba` devuelve ficheros y directorios. También habríamos podido utilizar `find . -type f` y ahorrarnos el `if` de la línea 35. Finalmente, tras terminar el bucle, si la variable `biggest`, que se ha inicializado a vacío en la línea 32 contiene algo será que lo ha encontrado en la línea 36, donde además se hace que el path sea absoluto mediante el comando `realpath`. En ese caso, se guarda el nombre (que se extrae encajando todos los caracteres que no son espacios al principio de línea) que se convierte mediante una sustitución de `sed` en el comando de salida (línea 42). Si `biggest` estaba vacío, el programa sale con error.

Esta estrategia de ordenar por tamaño para buscar el máximo es, a nivel algorítmico, nefasta, pero se utiliza mucho porque es muy cómoda de programar en la shell. La programamos así porque el listado de ficheros no será especialmente grande y programa está limitado por entrada salida. Si fuese un problema, se podría programar de forma un poco más avanzada, mediante `awk`:

```
 $ du -ba
8       ./rrr/a.txt
68      ./rrr
0       ./c
3       ./b
2       ./a
193     .
 $ du -ba | awk '{if(system("test -f "$2)==0){
                          if(max<$1) {
                                 max=$1;maxf=$2
                          }
                }}
        END{ print maxf }'
./rrr/a.txt
 $
```

2.30.2. Killusers

Escribe un script de shell llamado `killusers.sh` que reciba un conjunto de directorios como parámetro. El script matará todos los procesos que tengan el directorio y subdirectorios como directorios de trabajo. Adicionalmente, puede recibir un primer parámetro optativo (`-d`) para que en lugar de matar los procesos, escriba por su salida estándar el comando que ejecutaría. Si no hay ningún proceso que tenga esos directorios como directorio de trabajo, el programa no hará nada. Si alguno de los directorios que se le pasan no existe, debe salir con error y escribir `"error: dir XXXX not found"`, donde `XXX` es el nombre del directorio por su salida de error y no llegar a ejecutar ese parámetro ni parámetros posteriores. Ojo, hay que ser cuidadoso con el directorio actual en el script. El programa no debe suicidarse, ni matar ninguno de sus comandos (ni escribir comandos que lo hagan).

A continuación analizamos dos soluciones distintas, `killusers1.sh` y `killusers2.sh`:

Programa 2.22: killusers1.sh

```
#!/bin/sh


usage(){
    echo "usage: $0 [-d] dirs..." 1>&2
    exit 1
}

printonly=''
if [ $# -ge 1 ] && [ $1 = '-d' ]; then
    shift
    printonly='print'
fi

if [ $# = 0 ]; then
    usage
fi
dirs=''
for i in "$@";
do
    #para evitar /tmp/..
    cldir=$(realpath "$i")
    if [ "$cldir" = '/' ]; then
        echo "error $0: / is not a valid argument" 1>&2
        exit 1
    fi
    if ! [ -d "$cldir" ]; then
        echo "error $cldir is not a dir" 1>&2
        exit 1
    fi
    dirs="$cldir $dirs"
done

cd /

listpids=$(lsof +D $dirs|awk '$4 ~ /^cwd$/ {print $2}')

if [ "$listpids" = '' ]; then
    exit 0
fi

cmd=$(echo $listpids| sed 's/^/kill -KILL /g')

if [ "$printonly" = 'print' ]; then
    echo $cmd
else
    $cmd
fi
```

De forma similar al script anterior, primero se procesarán argumentos y luego se realizará el trabajo.

Procesado de argumentos

Primero se define la función de ayuda `usage` para salir en caso de argumentos erróneos.

A continuación se extrae el argumento opcional (líneas 10-12) asegurándose de que los argumentos contienen al menos un directorio(líneas 15-17). Se considera que si no se ha recibido ningún directorio es un error. El enunciado dice que, en caso de recibir un parámetro que no es un directorio, no se debe procesar ese parámetro ni ninguno posterior. Sin embargo, no dice nada de los parámetros correctos hasta ese momento. Interpretamos que se puede salir en la comprobación de argumentos al llegar a ese argumento sin hacer nada más, ignorando que haya parámetros correctos.

Hay que evitar que el programa se suicide. Una manera de hacerlo es prohibir que se pase `'/'` como argumento y que ese sea el directorio de trabajo del script. No tiene mucho sentido que se pasase eso como argumento, ya que mataría a todos los procesos que pudiese. Por esto podría ser una buena solución. Este directorio además, no puede desaparecer, ya que es el raíz de todo el sistema. Esta es la razón por la que los demonios lo suelen tener de directorio de trabajo.

Sin embargo, el directorio `'/'` se puede pasar al script indirectamente como un enlace simbólico o como `'/tmp/..'`. Para comprobar que esto no sucede, el script se recorre la lista de argumentos y tras ejecutar `realpath` sobre ellos, se comprueba si se corresponden con `'/'`, líneas 20-26). De paso se comprueba que los argumentos son, efectivamente, directorios (líneas 27-30). El comando `realpath` nos da un path absoluto limpio desde el raíz. De paso, se guardan los *paths* limpios para luego en la variable `dirs`.

La comprobación que se hace no evita que pueda haber un enlace duro al raíz. Para eso tendríamos que comparar el número de inodo, que es más avanzado. Si lo quisiésemos hacer de todas formas, habría que comparar el resultado de `ls -di | awk '{print $1}'` y si son iguales entrar en el `if` de la línea 23.

Lógica del script

El script se va al directorio raíz para protegerse y evitar matar ninguno de sus comandos (línea 34).

A continuación consigue la lista de identificadores de proceso (`pids`) de los procesos situados en los árboles que se le han pasado como parámetro al script. Para ello utiliza `lsof +D`. Esta opción no sigue enlaces simbólicos[32] (lo que hemos preferido en este caso, sin saber la topología, podría acabar en un bucle). El comando lsof imprime una línea por cada descriptor de fichero de cada proceso que tiene abierto el fichero o directorio. Cuando el nombre es `cwd`, (***c**urrent **w**orking **d**irectory*) significa que ése es su directorio de trabajo. En el caso de que la cuarta columna coincida exactamente con la string `cwd`, guardamos su `pid`, que es la segunda columna. Este filtrado lo realizamos mediante `awk`

[32]Los enlaces simbólicos están explicados en 4.3.3 .

(línea 36). Si no hay ningún `pid`, salimos sin error, como especifica el enunciado, (línea 38-40).

Finalmente se reeescribe la línea mediante `sed` para poner delante el comando `kill -KILL`. Este comando sirve para mandar señales a un proceso. Por defecto le manda la señal `TERM`. Esa señal le pide al proceso que salga ordenadamente. La señal `KILL` o 9 lo mata inmediatamente[33].

Finalmente, se recibió el modificador para imprimir el comando, se imprime, y si no, se ejecuta (líneas 44-48). Ojo, cuando se use el comando `test` o la variante que utilizamos aquí `[`, si uno de los lados es una variable que puede estar vacía, es importante utilizar comillas dobles para que no nos de el error de que le falta el primer argumento. No es lo mismo el argumento cadena vacía que ningún argumento.

Otra solución válida para el ejercicio podría ser `killusers2.sh`:

[33] Las señales están explicadas más en detalle en 6.4 .

Programa 2.23: killusers2.sh

```
#!/bin/sh

if test $# -gt 0 && test $1 = '-d'
then
    shift
    dflag=yes
fi

test $# -eq 0 && exit 0

dirs=''
for i in "$@"
do
    if ! test -d $i
    then
        echo error: dir $i not found >&2
        exit 1
    fi
    dir=$(realpath "$i")
    if [ $dir = '/' ]
    then
        echo error: / in params >&2
        exit 2
    fi
    dirs="$dir $dirs"
done

cd /
rexp=$(echo $dirs |sed -E 's/([^ a-zA-Z0-9''\''])/\\\1/g'| sed 's/ /|/g')
p=$(ls -l /proc/[0-9]*/cwd 2> /dev/null | \
        awk '{print $9 ":" $11}' | \
        egrep -- ":($rexp)" | \
        sed -E -e  's/^\/proc\///' -e 's/\/.*//')

if echo $p | egrep -q '[0-9]'
then
    if test $dflag
    then
        echo kill -9 $p
    else
        kill -9 $p
    fi
fi
```

Este script interpreta el enunciado de forma levemente diferente (en las partes que no especifica el enunciado) y además utiliza `/proc` directamente en lugar del comando `lsof`.

Procesado de argumentos

Primero extrae el argumento optativo (líneas 3-7) mediante `shift`. A continuación sale con éxito de forma silenciosa si no ha recibido argumentos, porque considera que no es un error. Por un lado, es una interpretación coherente con el enunciado, que especifica, que si no hay procesos que matar, debe el script salir con éxito. Por otro lado, no hacer nada de forma silenciosa, debe ser evitado en los programas en general para no ocultar al usuario que ha habido un error y hacer más fácil la depuración del programa. Dado el enunciado, ambas decisiones eran válidas y esta solución ha tomado la de salir de forma silenciosa. Es un compromiso razonable[34]. Es importante tomar estas decisiones de forma explícita y razonada, considerando los compromisos implicados, no elegir alguna sin darse cuenta por no haberlos ponderado.

Otra diferencia importante con el script anterior es que éste utiliza el comando `test` por ese nombre en lugar de `[`.

De la línea 12 a la 26 se comprueba que le han pasado directorios al script y se guardan en la variable `dirs` como en el script anterior.

Lógica del script

Primero se cambia al directorio raíz (línea 28) como en la otra solución. A continuación se construye una expresión regular que contiene todos los nombres de directorios como alternativa, es decir, si los argumentos son `/tmp/a` y `/tmp/b` construye la expresión regular `/tmp/a|tmp/b` pero antes se escapan todos los caracteres que no son espacios ni letras ni números ni comilla simple ni invertida en los nombres de los ficheros[35]. Escapar la comilla simple es interesante. Requiere cerrar las comillas, escapar las comilla y abrirlas de nuevo. Por ejemplo para escribir `x??'y??` por el terminal habría que ejecutar `echo 'x??'\''y??'` La expresión regular que hemos construido al vuelo queda, finalmente, similar a `\/tmp\/a|tmp\/b`. Escapar todos los caracteres especiales asegura que ninguno se interpreta como si fuese un caracter especial de una expresión regular. Por ejemplo, el fichero `/tmp/a.b` podría encajar como expresión regular con `/tmp/a/b`. Queremos evitar esto.

Esto lo hace la línea 29. Se nos podría pasar alguno (por ejemplo si está el carácter '\' o hay un espacio en el nombre). Como en el caso general de los espacios, hay un compromiso entre complicar demasiado el script (y se nos van a escapar seguro algunos caracteres) y tener nombres de ficheros y directorios razonables. La solución `killusers1.sh` es más robusta en este sentido. Esta solución, a cambio, no tiene dependencias extra, como el comando `lsof`. Estos compromisos hay que valorarlos en cada situación.

Una vez construida la expresión regular, buscamos los directorios de `/proc` que tienen

[34] Valorar compromisos, es decir en la solución de un problema multidimensional elegir a qué dimensiones se le dedican más recursos y a qué dimensiones menos, teniendo en cuenta que a veces mejorar una dimensión es empeorar otras, es parte fundamental del trabajo de un ingeniero. No hay solución perfecta: hay compromisos adecuados.

[35] Ojo con escapar letras o comillas, tiene un significado especial que no hemos visto aquí. Por ejemplo \w encaja con una palabra. Mira con detenimiento las expresiones regulares que implementa el comando si vas a hacer este tipo de cosas.

dentro un directorio `cwd` que es un enlace simbólico al directorio que buscamos. Por ejemplo, el proceso de `pid` 3223 tendría un enlace simbólico `cwd` apuntando a `/tmp/a`:

```
 $ cd 3223
 $ pwd
/proc/3223
 $ ls -l cwd
lrwxrwxrwx 1 paurea paurea 0 abr  8 20:49 /proc/3223/cwd -> /tmp/a
 $
```

En ese caso, ése es el que nos interesa. Para encontrarlo, se listan todos los directorios de `/proc` que se corresponden a procesos cuyo nombre es un `pid` y por tanto un número y dentro de ellos el directorio `cwd` utilizando globbing `/proc/[0-9]*/cwd`. Se redirecciona la salida de error a `/dev/null` para evitar los errores de permisos (línea 30). A continuación se extraen las columnas del nombre de directorio y del directorio de trabajo, separándolas por dos puntos, para luego recuperar el `pid`, (línea 31). Finalmente se filtra por la expresión regular que hemos construido antes (línea 32) y se sustitutye con `sed` dejar sólo el `pid` (línea 33).

Una vez obtenida la lista de identificadores de proceso que se guarda la variable `p`, se comprueba que no está vacía (si no, se acaba silenciosamente) y en tal caso se imprime el comando como en el script anterior, líneas 35-43. Nótese que en lugar de llamar a la señal por su nombre `KILL` se utiliza el número 9, que es equivalente.

2.30.3. Photosren

Escribe un script de shell llamado `photosren.sh` que reciba un directorio como parámetro. Tiene que buscar todos los ficheros que son imágenes y meterlos en un subdirectorio cuyo nombre será la fecha actual en un formato similar a `20_jan_2020`. Los nombres de los ficheros se renombrarán para que su nombre sea un número seguido de la extensión y todos los nombres sean de la misma longitud (sin incluir la extensión).

Para detectar qué ficheros son imágenes, se recomienda el uso del comando `file` (mirar *file(1)*).

El script debe recibir al menos un parámetro.

Por ejemplo:

```
 $   date
lun mar 23 10:05:22 CET 2020
 $   ls /tmp/x
a.gif
b.jpeg
c.txt
de.png
r.png
qwe.png
ss.gif
etc...
r
 $   photosren /tmp/x
 $   ls /tmp/x
23_mar_2020
c.txt
r
 $   ls /tmp/x/23_mar_2020
000.gif
001.jpeg
002.png
003.png
004.png
005.gif
etc...
103.gif
 $
```

Programa 2.24: photosren.sh

```
1  #!/bin/sh
2
3  usage(){
4      echo "usage: $0 dir" 1>&2
5      exit 1
6  }
7
8
9  if [ $# -ne 1 ] || ! [ -d "$1" ]; then
10     usage
11 fi
12
13
14 dirname=$1/$(date +%d_%b_%Y)
15
16 if [ -d "$dirname" ]; then
17     echo "error: dir $dirname already exists" 1>&2
18     exit 1
19 fi
20 mkdir $dirname
21
22 if ! [ -d "$dirname" ]; then
23         echo "error: dir $dirname could not be created" 1>&2
24         exit 1
25 fi
26
27 isimage(){
28     file --mime-type "$1"|sed 's/.*://g'|awk '{print $1}'|\
29         egrep '^image/'> /dev/null 2>&1
30 }
31
32 for i in "$1/"*; do
33         if isimage "$i"; then
34                 mv "$i" "$dirname/$i"
35         fi
36 done
37
38 nfile=$(ls $dirname/|wc -l)
39 orignames=$(ls $dirname/)
40
41 for i in $(seq -w 1 $nfile); do
42     fichname=$(echo $orignames|tr ' ' '\n'| sed "$i"q|tail -1)
43     ext=$(echo "$fichname"|sed -E 's/.*(\.[^.]+)$/\1/')
44     mv "$dirname/$fichname" "$dirname/$i$ext"
45 done
```

Este script comprueba que hay al menos un argumento y que es un directorio utilizando `test` en las líneas 8-10 y en ese caso llama a la función `usage` como los scripts anteriores. A continuación genera el nombre del directorio que va a crear. Para eso utiliza la cadena de formato de `date` (línea 13). Esta cadena es similar a la de `printf` y permite expresar

la fecha en el formato que queramos. En este caso, `%d` es el día numérico, `%b` es el mes abreviado y `%Y` es el año completo. Tras generar el nombre del directorio y meterlo en una variable, comprueba si ya existe y en tal caso da un error, líneas 15-18. El enunciado no especifica qué hacer en caso de que dicho directorio exista, pero puesto que vamos a crear ficheros dentro y podemos borrar datos importantes, es mejor dar un error. A continuación, se crea el directorio y si no es posible, se sale con error, líneas 19-24. Podríamos haber puesto como condición del `if` el `mkdir` directamente, pero así la comprobación es igual antes y después de crear el directorio.

A continuación, se define la función `isimage` para comprobar si un fichero es una imagen (líneas 27-30). Esta función, ejecuta `file --mime-type` sobre el fichero para buscar salidas de tipo `image/xxxxx`. Con `awk` y `egrep` filtramos si la línea contiene o no `image` en el lugar adecuado para saber si es una imagen. El estado de salida de ejecutar la función será el de ejecutar el último (y único) comando.

En caso de que sea una imagen, copiamos el fichero al directorio que hemos creado antes (líneas 32-36). Copiamos los ficheros para más adelante renombrarlos. De esta manera se hace más fácil la tarea de renombrar con números correlativos todos los ficheros del directorio.

Obsérvese que en la línea 32 se utiliza globbing para buscar los ficheros `$dirname/*`. Esto tiene como consecuencia que se ignoran los archivos ocultos (por ejemplo `.xx.png`). Se podría evitar diciendo explícitamente que se expandan los nombres que empiezan por punto, `$dirname/* $dirname/.*`. El enunciado no especifica nada, y hemos supuesto que el script no trabaja con ficheros ocultos. Es trivial hacer que lo haga. De nuevo es una decisión de diseño explícita. Ambas cosas son razonables y podrían tener su motivación.

Para la última tarea, primero se cuenta el número de ficheros que hay en el directorio que acabamos de crear y se guarda en una variable (línea 38) y los nombres de los ficheros en otra variable (línea 39). A continuación se itera en la lista 000, 001, 002... obtenida mediante el comando `seq -w 1 $nfile` que nos devuelve una lista de números desde 1 hasta el número de ficheros. Gracias al modificador `-w` el comando añade ceros para que todos los números sean del mismo ancho en caracteres. Se extrae el i-ésimo nombre de fichero (tras escribirlos uno por línea sustituyendo los espacios por finales de línea con `tr`) mediante `sed` y `tail` (línea 41) y se obtiene la extensión mediante `sed`, (línea 42). Finalmente se renombra el fichero (línea 43).

2.30.4. Notas

Escribe un script de shell llamado `notas.sh` que reciba un conjunto de ficheros como parámetro. En cada fichero habrá dos columnas, la primera será el DNI (con letra) y la segunda, las notas. El script creará un fichero `NOTAS` con la nota final (la media de todas las notas con un decimal) y `NP` si el alumno no ha presentado alguna de las notas (no está presente en alguno de los ficheros).

Adicionalmente, el script dará un error (y saldrá con el consiguiente estado) si cualquiera de las notas no está entre 0.0 y 10.0.

Por ejemplo:

```
 $ cat notas1
234234W 10.0
346456F 7.5
 $ cat nota2
234234W 1.0
 $ notas.sh notas1 notas2
 $ cat NOTAS
#dni nota
234234W 5.5
346456F NP
```

Programa 2.25: notas.sh

```
#!/bin/sh


usage(){
    echo "usage: $0 fich [fich ..]" 1>&2
    exit 1
}

badgrade(){
    echo "bad grade"
    exit 1
}


baddni(){
        echo "bad dni"
        exit 1
}
if [ $# -lt 1 ]; then
    usage
fi

if [ -e NOTAS ]; then
    echo "error: NOTAS already exists" 1>&2
    exit 1
fi

cat $@| awk '($2<0.0 || $2>10.0)|| ($2 !~ /\./ ) {exit (1);}' || badgrade

cat $@|awk '$1 !~ /^[0-9]+[a-zA-Z]$/ {exit(1);}'||baddni

names=$(cat $@|awk '{print $1}' | sort -u)

for i in $names; do
    nota=''
    nfiles=$(grep \^$i'[      ]' $@ /dev/null| \
        awk -F: '{print $1}'|sort -u|wc -l)
    if ! [ $nfiles = $# ]; then
        nota=NP
    fi
    if ! [ "$nota" = NP ]; then
        nota=$(grep $i $@| \
            awk '{avg+=$2}END{avg/=NR; printf("%2.1f", avg);}')
    fi
    echo "$i     $nota" >> NOTAS
done
```

Comprobación de argumentos y de la entrada

Como en los scripts anteriores, se comprueban primero los argumentos. Primero, se definen unas cuantas funciones para procesar errores (líneas 4-18). A continuación, se comprueba que al menos hay un argumento (líneas 19-21) y que no existe previamente el fichero `NOTAS` que crea el script.

El *pipeline* de la línea 28 concatena todos los ficheros de la entrada y comprueba mediante `awk` que la segunda columna es una nota válida. Hace dos comprobaciones, que es un número entre 0.0 y 10.0 y que contiene un punto. Hay que recordar que `awk` es de tipado débil y está pensado justo para estos casos. La primera comprobación será falsa si en la entrada hay un número y además está entre 0 y 10. Es decir, será verdadera para la cadena `"hola"` y falsa tanto para 4 como para 4.0

No comprueba que el número tenga que estar en coma flotante (técnicamente, no nos especificaba nada sobre esto el enunciado). La segunda comprueba que tiene al menos un punto. El script aun así tolera entradas como $1.1e-3$. Se podría hacer más precisa la expresión regular con algo como `(10\.0)|([0-9]\.[0-9]+)` o forzar un número concreto de dígitos. El enunciado no especificaba nada y hemos optado por comprobar algo sencillo.

El segundo *pipeline*, de la línea 30 comprueba mediante una expresión regular que la columna del DNI sea un conjunto de números y una letra al final.

Obsérvese que el programa de `awk` llama en ambos casos a la función `exit` para que salga con error, por tanto el *pipeline* salga con error y se llame a la función adecuada en ese caso (utilizando el *cortocircuito* del builtin `||`).

En la línea 32 se extraen los DNIs de los usuarios, que se utilizan como claves. Después mediante `sort -u` se ordenan y se quitan los repetidos. Es equivalente a `sort|uniq`. Ordenar un conjunto de elementos para extraer o calcular las repeticiones es una estrategia extremadamente común y algorítmicamente eficiente.

Finalmente, se busca cada DNI en todos los ficheros de forma precisa, es decir, debe ir desde el principio del línea hasta el tabulador. Esto es así porque podría haber dos DNIs que fuesen uno un sufijo del otro (por ejemplo 316484G y 4G son ambos DNIs válidos). A grep se le pasa como fichero adicional `/dev/null` para que imprima el nombre del fichero (si recibe un sólo nombre no lo imprime), se extraen los nombres de ficheros, se ordenan, se quitan repeticiones y se cuentan con `wc` (líneas 36-37). Si el número de ficheros no se corresponde con los que se han recibido como parámetro, se guarda en `nota` la cadena `NP` (líneas 38-40). Finalmente, si la nota no es `NP` se calcula la media mediante `awk` sumando todos los valores y dividiendo al final entre el número de registros (líneas 42-42). Además se imprime con la cadena de formato de `printf` para números en coma flotante y pidiendo dos cifras significativas y un decimal `%2.1f`. Una vez calculada la cadena de nota, se concatena al final del fichero mediante $\gg$, en la línea 45.

La forma de calcular la media de esta solución es razonable cuando hay pocos números, pero es peligrosa por la posibilidad de desbordamiento y la acumulación del error, en especial si hay una cantidad grande de números a la entrada. Para evitar esos problemas, se puede utilizar un algoritmo *online* (sobre la marcha) para calcular la media:

```
 $ echo '1.0
 2.0
 3.0
 4.0
 5.0' | awk awk 'BEGIN{avg = 0.0}
         {avg += ($2 - avg)/NR;}
         END{printf("%2.1f\", avg);}'
3.0
 $
```

2.30.5. Catletter

Escribe un script de shell llamado `catletter.sh` que, para todos los ficheros con nombre acabado en `.txt` del directorio que se pasa como único argumento del script, concatene el contenido de los ficheros a otros ficheros. Los ficheros cuyo nombre comience con un carácter `x`, deben concatenarse al fichero `x.output`.

Se debe mantener un orden alfanumérico en base al nombre completo del fichero para realizar las concatenaciones.

Mayúscula y minúscula se deben considerar iguales a la hora de elegir el fichero de salida (el fichero de salida se llamará con el carácter en minúscula).

Si los ficheros de salida (.output) existen, se deben borrar antes de la generación de los nuevos ficheros de salida. El borrado debe ser silencioso (no se deben escribir mensajes de error por el borrado de dichos ficheros).

Por ejemplo, supongamos que el directorio contiene únicamente dos ficheros que empiezan con `'c'`, que son `Carta.txt` y `contrato.txt`. El contenido del fichero resultante `c.output` será el contenido de los ficheros `Carta.txt` y `contrato.txt` (en ese orden).

Un ejemplo de ejecución:

```
 $   echo uno dos tres > cometa.txt
 $   echo one two three > Camion.txt
 $   echo hola hola > cohete.txt
 $   echo deadbeef > Avion.txt
 $   ./catletter.sh .
 $   ls *.output
a.output  c.output
 $   cat a.output
deadbeef
 $   cat c.output
one two three
hola hola
uno dos tres
 $
```

Programa 2.26: catletter.sh

```
#!/bin/sh

usage(){
    echo "usage: $0 dir" 1>&2
    exit 1
}
if [ $# -ne 1 ] || ! [ -d "$1" ]; then
        usage
fi

dirname=$1
cd $dirname
if ls *.txt > /dev/null 2>&1; then
    echo "$0: no *.txt files"
    exit 1
fi

ls *.txt|sed -E 's/((.).*\.txt$)/rm -f \2.output/'| \
    awk '{print $1, $2, tolower($3)}'|sort -u|sh
ls *.txt|sort| \
    sed -E 's/((.).*\.txt$)/cat \1>> \2.output/'| \
    awk '{print $1, $2, tolower($3)}'|sh
```

Este ejercicio es muy similar a los que ya se han corregido. Por ello, lo hemos programado de una forma un poco diferente para ilustrar el patrón de programación típico de la shell de escribir comandos como parte de un *pipeline* que se vio en la sección 2.22.

Procesado de argumentos

Es muy similar a los de los scripts anteriores. Primero comprueba que hay al menos un argumento y que es un directorio (líneas 7-9), se traslada a ese directorio (línea 11-12) y comprueba que hay algún fichero acabado en `.txt` dentro (líneas 13-16). Ignora los ficheros ocultos (que empiezan por punto) como consecuencia de expandir el patrón de globbing $*$.

Lógica del script

El primer *pipeline* (líneas 18-19) borra los ficheros de salida. Para ello escribe un conjunto de comando del tipo `rm -f x.output` para cada fichero de entrada. Primero `ls *.txt` imprime una lista de ficheros, por ejemplo `a.txt A.txt d.txt ab.txt` uno por línea. A continuación `sed` hace la sustitución mediante *backreferences* y sustituye estos ficheros por `rm -f a.txt`, `rm -f A.txt`, `d.txt` y `rm -f a.txt`. El comando `awk` pasa a minúsculas el segundo argumento (el nombre de fichero) mediante la función `tolower`. Podría haberse hecho de forma equivalemnte con `tr A-Z a-z`. Finalmente, `sort -u` ordena y quita los repetidos y `sh` ejecuta los comandos.

El segundo *pipeline* (líneas 20-22) hace algo parecido al primero pero ejecutando comandos similares a `cat ab.txt >> a.output` ordenados lexicográficamente según el orden de los ficheros de entrada (el primer `sort` del *pipeline*).

2.30.6. Waitexit

Escribe un script de shell llamado `waitexit.sh` que reciba como parámetro obligatorio un nombre de comando y una lista de *pids* (al menos uno). Se debe quedar esperando (haciendo *polling* cada segundo) hasta que todos los procesos asociados a los pids que se han pasado como parámetro salgan en cuyo caso debe ejecutar el comando. Si han pasado 20 segundos y no han salido, debe matarlos y no ejecutar el comando. Para matarlos hay que mandarles la señal `-KILL` (ver *kill(1)*).

Una solución se puede ver a continuación:

Programa 2.27: waitexit.sh

```
1  #!/bin/sh
2
3
4  usage(){
5      echo "usage: $0 cmd pid [pid ..]" 1>&2
6      exit 1
7  }
8
9  if [ $# -le 1 ] || ! which "$1" >/dev/null; then
10     usage
11 fi
12
13 cmd="$1"
14 shift
15 if ! echo $*|egrep '[0-9 ]'>/dev/null 2>&1; then
16     usage
17 fi
18
19 regexp=$(echo $*| sed -E -e 's/[     ]+/|/g' -e  's/^/\^/' -e 's/$/\$/')
20
21 finish=some
22 for i in $(seq 1 20); do
23     if ! ps aux| awk '{print $2}' |egrep "$regexp">/dev/null 2>&1; then
24         finish=noneexec
25         break
26     fi
27     sleep 1
28 done
29
30 if [ $finish = some ]; then
31     kill -KILL $* 2> /dev/null
32     exit 0
33 fi
34
35 $cmd
```

El procesado de argumentos es similar a los scripts anteriores. Una cosa reseñable es el uso de `which` para asegurarse de que el primer argumento es un comando (línea 9).

Como se comprueba que todos los argumentos restantes son números o espacios (línea 15), se puede construir sin peligro una expresión regular, sustituyendo cualquier número de espacios o tabuladores por el operador alternativa | y encajando final y principio de línea.

Así, si el script recibe como parámetro `waitexit.sh ls 234 325 34` se construye la expresión regular ^234|325|34$. Es importante poner delimitadores a la expresión regular para no confundir números que están incluidos unos en otros, por ejemplo, 23 y 42355.

En el bucle de polling, que da 20 vueltas (líneas 22-28), si no hay ninguna línea de `ps aux` la segunda columna (el `pid`) que encaje con la expresión regular. En caso de que encaje, se apunta en la variable `finish` y se sale del bucle con `break` (línea 23-26). En

caso contrario, se espera un segundo a la siguiente vuelta (línea 27). Si el bucle no ha salido a causa del `break` se matan los procesos con `kill` y se sale (líneas 30-33). En caso contrario, se ejecuta el comando (línea 35).

2.31. Ejercicios no resueltos

1. Escribe un script que lea una lista de directorios de un fichero de configuración y devuelva los últimos 5 ficheros que se han creado en esos directorios.†∗

2. Escribe un script que dado un fichero con rutas te diga la profundidad máxima de las mismas. Pista: puedes contar `/`.

3. Escribe un script que dado un árbol de directorios, lo aplane (deje todos los ficheros a todos los niveles en un sólo directorio de nombre `flat_dir` en el raíz del directorio.†∗

4. Escribe un script que haga exactamente lo mismo que `find . -ls -type f` utilizando `du` y `ls`.†∗

5. Escribe un script que calcule la letra de un DNI. Escribe otro que la compruebe. Los detalles de cómo calcularlo están en:

   http://www.interior.gob.es/web/servicios-al-ciudadano/dni/calculo-del-digito-de-control-del-nif-nie.
   Pista: ésta puede ser una tarea más fácil de escribir en `awk`. †∗

6. Escribe un script que se baje una página web cada minuto y te avise cuando aparece un conjunto de palabras que recibe como parámetro (o cuando ha cambiado si no te pasan parámetros). Pista: utiliza `wget`.†∗

7. Escribe un script que renombre todos los ficheros de un directorio para sustituir los espacios por _. Escríbelo varias veces, una utilizando `sed`, otra `awk` y otra `tr` para las sustituciones.

8. Escribe un script que cambie los fines de línea de Windows a Unix `\r\n` a `\n` y otro que haga lo contrario. Pista: utiliza `tr` y `sed`. Comprueba el resultado con `xd -c -x1`. † ∗ ∗

9. Escribe un script que dado un nombre de fichero en un repositorio de git, imprima el comando para verlo en todos los commits que tengan tag. Pista: utiliza `git ls-tree`, `git cat-file` y `git tag`.†∗

10. Escribe un script que dados dos números enteros positivos (el segundo mayor que el primero) y un ancho escriba todo el intervalo de números con tantos dígitos (poniendo ceros delante) como dicte el ancho.†∗

11. Rescribe el script de ojos rojos de `gimp` en Python † † ∗∗. Escribe un script de shell que ejecute uno de los dos scripts (el de script-fu o el de Python) con diferentes valores de umbral y genere ficheros con nombres diferentes para cada número †∗.

3 Procesos

3.1. Programas, bibliotecas y llamadas al sistema

Como ya sabemos, un programador escribe un programa en código fuente, lo compila y enlaza para tener un programa binario. Para representar un programa en ejecución, el sistema operativo ofrece la abstracción de *proceso*. Un proceso es una abstracción que representa un flujo de ejecución. Nos permite escribir programas sin tener que preocuparnos de que hay otros programas en ejecución a la vez que el nuestro. Un proceso interactúa con el sistema como si fuese el único programa en ejecución. Se puede programar y enlazar como si toda la memoria fuese suya, toda la CPU fuese suya, etc. Por supuesto, todo esto es una invención del sistema.

Al final el código que ejecuta un programa viene de:

- El código que escribe el programador.
- Las bibliotecas a las que llama el programa (o las propias bibliotecas).
- Las llamadas al sistema que efectúa el programa (o las bibliotecas a las que llama).

Siempre que escribimos un programa en C se enlaza con la biblioteca estándar de C, la `libc`. En un sistema GNU/Linux, la versión de esa biblioteca es `glibc` *(GNU libc)*. No hace falta especificar nada a la hora de invocar al compilador o el enlazador, se enlaza con esa biblioteca automáticamente.

Las llamadas al sistema se realizan llamando a una función *stub* de la libc. Esto significa que, por ejemplo, para realizar la llamada al sistema `open`, debemos llamar a una función de la libc que se llama `open`[1]. Esa función hace lo necesario para realizar la llamada al sistema como se explicó en el Capítulo 1. A esa pequeña función de la libc se le llama *stub* de la llamada al sistema.

Tenemos dos comandos que nos permiten trazar qué llamadas al sistema y llamadas a biblioteca[2] se realizan cuando se ejecuta un programa.

Para lo primero tenemos el comando `strace`. Para lo segundo, el comando `ltrace`.

Por ejemplo, supongamos este sencillo programa:

Programa 3.1: hello.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int
5 main(int argc, char *argv[])
6 {
7     printf("hello world!\n");
8     exit(EXIT_SUCCESS);
9 }
```

Cuando se ejecuta el programa se realizan las siguientes llamadas al sistema y llamadas a biblioteca:

[1] La descrita en *open(2)*.

[2] En realidad sólo las bibliotecas dinámicas.

```
 $ ./hello
hello world!
 $ strace ./hello 2>&1 >/dev/null | sed -E 's/[()].*//'
execve
brk
access
access
openat
fstat
mmap
close
access
openat
read
fstat
mmap
mmap
mprotect
mmap
mmap
close
arch_prctl
mprotect
mprotect
mprotect
munmap
fstat
ioctl
brk
brk
write
exit_group
+++ exited with 0 +++
$ ltrace ./hello >/dev/null
puts("hello world!")
exit(0 <no return ...>
+++ exited (status 0) +++
$
```

Como vemos, incluso para ejecutar un programa tan sencillo como éste se realizan una gran cantidad de llamadas al sistema. El *pipeline* del ejemplo sirve para mostrar únicamente el nombre de la llamada realizada.

También podemos ver que no se hace una llamada a la función *printf(3)*. El compilador gcc cambia las llamadas a `printf` por llamadas a *puts(3)* cuando puede, es decir, cuando

la string no contiene formato, por cuestiones de optimización[3]. Es lo que ha pasado en este caso.

Hay funciones de biblioteca que son *thread safe* para cuando usemos hilos (algunas tienen dos versiones, una que lo es y otra que no). Siempre debemos intentar usar funciones *thread safe* por si en algún momento decidimos hacer nuestro programa multihilo (si el programa es multihilo, **sólo** debe usar funciones *thread safe*). En el Capítulo 7 hablaremos más sobre esto.

3.1.1. Manejo de errores de llamadas

En general, cuando usamos funciones de biblioteca, siempre debemos comprobar si ha habido un error. Por supuesto, con las llamadas al sistema ocurre lo mismo. Las llamadas al sistema suelen retornar un valor negativo en caso de error (`NULL` si retornan un puntero), pero debemos consultar la página de manual correspondiente para estar seguros de que estamos comprobando los errores correctamente.

La libc mantiene una variable global[4] llamada *errno(3)* en la que se almacena un código que describe el error al retornar de una llamada al sistema o función de biblioteca. Hay funciones que no actualizan `errno`. Ese tipo de detalles (si se usa `errno`, qué valor se asigna en cada tipo de error, etc.) se describen en la página de manual de cada función[5]. En particular, es interesante ver la página de manual *intro(2)* y *errno(3)*. Los distintos valores de `errno` tienen cadenas de texto asociadas.

Tenemos distintas funciones para manejar los errores:

- `perror`: imprime la cadena asociada a `errno`. Si se pasa como argumento una cadena, escribe primero la cadena y después el error.

  ```
  void perror(const char *s);
  ```

- `strerror`: devuelve la cadena asociada al error que se le pasa. La función `strerror_r` es la versión *thread safe* y requiere un array en el que meter la string (`strerror` devuelve una cadena de sólo lectura que pueden modificar llamadas subsiguientes).

  ```
  char * strerror(int errnum);
  ```

- `warn`: escribe el error de la última llamada concatenado después de la cadena con formato que se le pasa.

  ```
  void warn(const char *fmt, ...);
  ```

3.2. Programas y procesos

Ya sabemos que para crear un fichero ejecutable hay que compilar los ficheros fuente y enlazar los ficheros objeto obtenidos. Durante el proceso de enlazado se realiza la *resolución de símbolos*. Al compilar, si el fuente usa funciones y variables que se encuentran

[3] Podemos pasar modificadores al compilador para evitar la optimización.

[4] En realidad hay una de estas variables por thread (en *pthreads(7)*). Veremos más en el tema 7.5.1 .

[5] También están los *includes* que hay que poner en el programa en C para poder utilizarlas y en qué orden.

definidas en otros ficheros fuente, todo eso quedará anotado en la tabla de símbolos del fichero objeto. Todos estos símbolos se tienen que resolver en el enlazado. Elegir una dirección para un símbolo (o en el caso de enlazado dinámico, que veremos a continuación, comprobar que existe el símbolo para poder elegir una dirección en tiempo de ejecución) se llama *resolver el símbolo*. Simplificando mucho, esto requiere elegir dónde se situarán en la memoria RAM, cuando se encuentre en ejecución, las diferentes partes del programa. Como consecuencia, cuando se usa una variable en el texto (con una instrucción tipo `LOAD`, `STORE` o `MOV`) o se salta a una subrutina (con una instrucción tipo `CALL`), el enlazador puede saber a qué dirección corresponde y escribirlo en el ejecutable. Este proceso es mucho mas sencillo gracias a que el sistema operativo simula para el proceso, con ayuda del hardware, que la memoria RAM es siempre igual, empieza en cero y es contigua (memoria virtual, explicada en la sección 5.3).

Una vez resueltos los símbolos, se puede generar el fichero ejecutable. El fichero ejecutable resultante no es más que una *receta* para el sistema operativo[6] de cómo crear un proceso que ejecute ese programa.

El enlazado de un programa puede ser:

- **Estático**: cuando se genera el binario ejecutable, se incluye todo el código necesario en el binario (el del programa y el código que usa de todas las bibliotecas). Los símbolos se resuelven a una dirección concreta (que puede ser relativa a una dirección de comienzo del programa que se elige en ejecución). Para ejecutar el programa sólo se necesita el fichero binario.

- **Dinámico**: el binario no incluye el código de las bibliotecas. En *tiempo de ejecución*, el sistema operativo carga/enlaza las bibliotecas que intenta usar el binario. En el proceso de enlazado del fichero no se resuelve totalmente el símbolo, solo se comprueba y se anota en el ejecutable que dicho símbolo existe en el fichero objeto de la biblioteca dinámica a la que pertenece. Por tanto, para ejecutar el programa necesitamos tanto el fichero ejecutable, como todas las bibliotecas dinámicas de las que depende y el enlazador dinámico.

  Se dice que son **bibliotecas compartidas** (*shared libraries*) cuando los distintos procesos que usan una misma biblioteca dinámica la comparten tanto en almacenamiento como en memoria[19, Capítulo 7.7]. Es la implementación habitual de bibliotecas dinámicas.

Normalmente se usa una combinación de los dos tipos. El enlazado de los diferentes objetos del propio programa (que no se encuentran en bibliotecas) suele ser estático. Para las bibliotecas externas, actualmente se usa más el enlazado dinámico que el estático.[7]

[6] En concreto para el cargador: el componente del sistema operativo que se ocupa de ejecutar un programa.

[7] En linux las bibliotecas dinámicas contienen *.so* (**S**hared **O**bject) en el nombre y las estáticas acaban en *.a*. Las bibliotecas estáticas no son más que un fichero tipo *ar*, un comando similar a *tar* que se utiliza para aglutinar un conjunto de ficheros objeto *.o*. Las bibliotecas dinámicas se generan pasándole a `gcc` en el enlazado el parámetro `-shared`. Son un único objeto de tipo ELF.

Ambos tipos tienen pros y contras. Si es estático, hay menos dependencias y la ejecución es más rápida, pero los ejecutables son más grandes. Si es dinámico, podemos ahorrar espacio. Otra ventaja de compartir el código de las bibliotecas es un mejor uso de las cachés del procesador. Sin embargo, pueden aparecer problemas de configuración con distintas versiones de la misma biblioteca[8]. Además, las bibliotecas dinámicas añaden complejidad y problemas de seguridad[9].

En general, por omisión se realiza enlazado dinámico (incluso puede ser un quebradero de cabeza intentar enlazar un programa estáticamente en sistemas como Linux, porque aprovecha el enlazado dinámico para cachés de resolución de nombres y otras cosas).

Una vez compilado y enlazado el programa, tendremos un fichero binario ejecutable. El fichero binario contiene la información para crear un proceso, organizada en secciones. Hay distintos formatos de ficheros ejecutables. En los sistemas de tipo Unix modernos se usa el formato *ELF* (***E**xtensible **L**inking **F**ormat*). Otros sistemas operativos usan otros formatos: PE (Windows), a.out (sistemas Unix antiguos), Mach-O (Mac OS X, iOS), etc.

En realidad, el formato ELF se usa para describir cuatro tipos distintos de ficheros: ficheros objeto, bibliotecas compartidas, binarios ejecutables[10] y ficheros *core* (para depurar). Un fichero ELF tiene una cabecera con datos fundamentales como la arquitectura o sistema para el que se ha generado el binario. Esta es la cabecera de ELF para sistemas de 64 bits, expresada como una estructura de C:

```
1  typedef struct {
2    unsigned char e_ident[EI_NIDENT];     /* Magic, endianess, etc.    */
3    Elf64_Half    e_type;          /* Object file type                  */
4    Elf64_Half    e_machine;       /* Architecture                      */
5    Elf64_Word    e_version;       /* Object file version               */
6    Elf64_Addr    e_entry;         /* Entry point virtual address       */
7    Elf64_Off     e_phoff;         /* Program header table file offset  */
8    Elf64_Off     e_shoff;         /* Section header table file offset  */
9    Elf64_Word    e_flags;         /* Processor-specific flags          */
10   Elf64_Half    e_ehsize;        /* ELF header size in bytes          */
11   Elf64_Half    e_phentsize;     /* Program header table entry size   */
12   Elf64_Half    e_phnum;         /* Program header table entry count  */
13   Elf64_Half    e_shentsize;     /* Section header table entry size   */
14   Elf64_Half    e_shnum;         /* Section header table entry count  */
15   Elf64_Half    e_shstrndx;      /* Section header string table index */
16 } Elf64_Ehdr;
```

Como se puede ver, la cabecera tiene cosas como el *magic number* (la secuencia de bytes que identifica el fichero como un ELF), el tipo de ELF, el orden (*endianess*) de los bytes (por ejemplo, *LSB* significa ***L**ess **S**ignificant **B**it*, o sea, *little-endian*), tipo de

[8] Lo que se llama *DLL hell*. Las DLLs son las bibliotecas dinámicas de los sistemas Windows.

[9] Por ejemplo, que se reemplace una biblioteca legítima por una biblioteca maliciosa (*hijacking/hooking*) o que se modifiquen las estructuras necesarias para el enlazado con el fin de secuestrar el flujo de ejecución.

[10] Algunos *ejecutables* pueden ser en realidad ELF del tipo *shared object*. Esto pasa para los binarios compilados como *PIE* (***P**osition **I**ndependent **E**xecutable*). Esto es muy habitual en sistemas modernos.

arquitectura, etc. La cabecera también indica la posición y tamaño de tablas que serán necesarias para el enlazado y la carga del programa.

El punto de entrada (*entry point*) es la dirección de memoria donde está la primera instrucción del programa. En general, **no** es la dirección de memoria donde está la función `main`. El punto de entrada es una rutina del *runtime* de C, que acaba llamando a la función `main` de tu programa. En la glibc, esa rutina se llama `_start`. Esta rutina es la que prepara lo que le hace falta a C para ejecutar: traduce entre la interfaz binaria del sistema operativo (*ABI*, ***A**plication **B**inary **I**nterface*) y la interfaz de un programa en C (esto es, `main` y sus parámetros).

El fichero ELF tiene dos tablas importantes:

- **Section Header Table (SH)**: describe las **secciones** del fichero ELF (donde empiezan en el fichero y su tamaño). Las secciones tienen la información necesaria para el enlazado del programa, esto es, necesaria en *tiempo de enlazado*.

  Algunas secciones importantes son:

  - `.text`: código (instrucciones) del fichero.
  - `.data, .rodata`: valores iniciales para las variables inicializadas y descripción de las mismas.
  - `.bss`: descripción de las variables sin inicializar.
  - `.plt, .got, .got.plt, .dynamic`: información para el enlazado dinámico.
  - `.symtab, .dynsym`: es la **tabla de símbolos**, que es información sobre los identificadores (variables, nombres de función, etc.), útil para depurar los programas.
  - `.strtab, .dynstr`: tabla con las cadenas de texto de la tabla de símbolos y las secciones (nombres de funciones, variables, etc.).

- **Program Header Table (PH)**: describe los **segmentos** del proceso que ejecuta el binario. Los segmentos son la abstracción del sistema operativo para las diferentes partes de la memoria de un proceso (variables globales, pila, etc.). El fichero ELF describe qué partes habrá y con qué valores se cargarán. Estos datos son necesarios en *tiempo de ejecución*.

  Un segmento puede corresponderse con cero o más secciones. Tiene atributos (solo-lectura, lectura/escritura, ejecutable), indica dónde se tiene que cargar en memoria, alineación, etc. Hay varios tipos de segmentos, por ejemplo:

  - `PT_LOAD`: es el tipo más importante. Define un segmento que se tiene que *cargar* en la memoria del proceso, esto es, se debe copiar del fichero ELF a la memoria.
  - `PT_INTERP`: indica la ruta del *enlazador dinámico*, que veremos más adelante.

- `PT_NOTE`: suele tener información adicional sobre el creador del programa, etc. Normalmente no contiene información necesaria para la ejecución, pero hay casos en los que sí[11].

Como hemos dicho, las secciones no son necesarias en tiempo de ejecución. Por tanto, pueden ser eliminadas del binario ejecutable después del enlazado. Por ejemplo, el enlazador usa la **tabla de símbolos** para resolver las direcciones de cada objeto (p. ej. en qué dirección de memoria estará una variable global). Para cada símbolo se guarda: nombre, tamaño, ámbito (local, global), tipo (función, variable, etc.), etc. Una vez que está enlazado el programa, ya no se necesita la tabla de símbolos para ejecutar el binario. Sin embargo, los depuradores también usan la tabla de símbolos porque facilita la depuración. En cualquier caso, la tabla de símbolos **no es necesaria** para ejecutar un programa.

El comando `strip` elimina la tabla de símbolos de un fichero. Podemos ver los símbolos de un ELF con el comando `nm`. Por ejemplo:

[11] Por ejemplo, los programas escritos en *Go* sí hacen uso de la información de esos segmentos.

```
 $ nm hello
0000000000201010 B __bss_start
0000000000201010 b completed.7698
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000201000 D __data_start
0000000000201000 W data_start
00000000000005b0 t deregister_tm_clones
0000000000000640 t __do_global_dtors_aux
0000000000200db8 t __do_global_dtors_aux_fini_array_entry
0000000000201008 D __dso_handle
0000000000200dc0 d _DYNAMIC
0000000000201010 D _edata
0000000000201018 B _end
                 U exit@@GLIBC_2.2.5
0000000000000724 T _fini
0000000000000680 t frame_dummy
0000000000200db0 t __frame_dummy_init_array_entry
0000000000000884 r __FRAME_END__
0000000000200fb0 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000000744 r __GNU_EH_FRAME_HDR
0000000000000520 T _init
0000000000200db8 t __init_array_end
0000000000200db0 t __init_array_start
0000000000000730 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000000720 T __libc_csu_fini
00000000000006b0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
000000000000068a T main
                 U puts@@GLIBC_2.2.5
00000000000005f0 t register_tm_clones
0000000000000580 T _start
0000000000201010 D __TMC_END__
 $ readelf --sections hello | grep sym
  [ 6] .dynsym           DYNSYM           00000000000003c8  000003c8
  [28] .symtab           SYMTAB           0000000000000000  00003040
 $ ls -l hello
-rwxrwxr-x 1 esoriano esoriano 8344 abr 21 14:11 hello
 $ strip hello
 $ nm hello
nm: hello: no symbols
 $ ls -l hello
-rwxrwxr-x 1 esoriano esoriano 6120 abr 21 16:42 hello
 $ readelf --sections hello | grep sym
  [ 6] .dynsym           DYNSYM           00000000000003c8  000003c8
 $
```

Se puede ver que el fichero, después de quitar la tabla de símbolos, ocupa unos 2 KB menos.

El comando `readelf` sirve para inspeccionar las partes de un fichero ELF. Podemos ver que, después de ejecutar `strip`, la sección correspondiente a la tabla de símbolos ha desaparecido.

El comando `nm` muestra los símbolos de un fichero objeto o ejecutable, y la posición relativa que tendrán en memoria cuando se ejecute el programa. Por ejemplo, algunos de los tipos de símbolos (si la letra es minúscula, es un símbolo local, en otro caso es

externo) reportados por `nm` son:

- `T`: indica que está en el segmento de texto, donde hay instrucciones (*text*). Son subprogramas, etiquetas, etc.
- `D`: indica que está en el segmento de datos (*data*). Normalmente variables globales inicializadas.
- `R`: indica que es una variable de solo-lectura (*read-only*). Por ejemplo, literales de tipo string.
- `B`: indica que está en el segmento de datos sin inicializar (*BSS*). Normalmente variables globales sin inicializar.
- `U`: indica que es un símbolo que está pendiente de resolver (*undefined*). Ese símbolo es externo, y todavía no se sabe dónde estará. Se resolverá al ejecutar el programa.

Simplificando, cuando se ejecuta un fichero, un componente del kernel llamado **cargador**[12] realiza las siguientes acciones:

1. Comprueba si el fichero se puede ejecutar (permisos, etc.).
2. Comprueba el tipo de ejecutable, mirando su número mágico (*magic number*). Si el fichero empieza por los caracteres `#!`, se trata de un programa interpretado: se ejecutará el interprete indicado a continuación, con la ruta de este fichero como argumento como vimos en el Capítulo 2. Si es un ELF, continúa el proceso.
3. Copia las partes debidas del ELF a la memoria, que como ya hemos visto, se organiza en segmentos[13].
4. Copia los argumentos del programa y variables de entorno (`argc`, `argv`, `envp`) a la pila de usuario del proceso.
5. Inicializa el contexto del proceso (registros, etc.) y sus atributos. Entre ellos, se pone el puntero de pila (SP) y el contador de programa (PC) como corresponda.
6. El proceso se *pone a ejecutar*. Al final, el proceso comenzará a ejecutar las instrucciones del ELF desde su *punto de entrada*. Si el ELF tiene segmento `PT_INTERP` (esto es, si se hace enlazado dinámico), antes se ejecutará el *enlazador dinámico*.

En general, la representación de la memoria de un proceso en arquitecturas x86/AMD64 se muestra en la Figura 3.1. Esta figura está simplificada y puede haber más de un mapa o región de memoria correspondiente a cada *segmento* del diagrama.

Las bibliotecas dinámicas compartidas se proyectan en regiones de la memoria situadas entre el BSS y la pila. La memoria dinámica se obtiene de una zona al final del BSS que se denomina *heap*.

[12] No confundir con el cargador de arranque (*boot loader*), que realiza una tarea similar, pero para cargar el kernel.

[13] Esta es una descripción simplificada. Ya veremos el capítulo dedicado a la gestión de la memoria que esto no es tan sencillo, ya que se usan estrategias perezosas como la *paginación en demanda*.

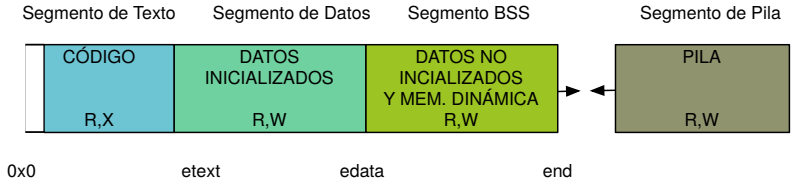

Figura 3.1: Representación de los segmentos de un proceso.

3.3. Enlazado dinámico

Cuando el enlazado es dinámico, el **enlazador dinámico** debe resolver los símbolos que están sin definir en el ELF en tiempo de ejecución. Los símbolos que muestra `nm` como *undefined* son los que están por resolver. En Linux, el enlazador dinámico es `/lib/ld-linux.so`.

Básicamente, una relocalización es esto: *"reemplaza el valor de X bytes que está en el offset Y por la dirección del símbolo externo S"*, es decir, sustituye el uso del símbolo por la dirección adecuada (último paso de resolución del símbolo).

Se puede forzar a que se resuelvan todos los símbolos al principio de la ejecución; el enlazador tiene opciones para ello. También se puede indicar que las regiones de la memoria que contienen las estructuras que veremos a continuación sean de sólo lectura. Esto se llama *RELRO* (***Rel**ocation **R**ead-**O**nly*) y se hace principalmente como medida de seguridad[14].

En general, se solía hacer **lazy binding**: la resolución no se hace de golpe antes de comenzar a ejecutar el binario, sino que se hace a medida que se van accediendo a los símbolos. Esta es una estrategia perezosa. Se retrasa la vinculación hasta que es inevitable hacerla. Cada vez es más común tener activado RELRO por motivos de seguridad.

Hay dos estructuras que se usan para implementar el enlazado dinámico: *GOT* y *PLT*. La *GOT* (***G**lobal **O**ffset **T**able*) es una tabla que contiene las direcciones de los símbolos, esto es, la dirección a la que tiene que saltar: un *trampolín*. La *PLT* (***P**rocedure **L**inkage **T**able*) contiene el código de los *trampolines* (o stubs).

Las relocalizaciones se realizan en la GOT, no en el texto (código), que es de sólo lectura. Cuando se resuelve un símbolo, se *parchea* la GOT: se escribe la dirección en la entrada correspondiente de la tabla. El código siempre llama a la entrada en la PLT para la función a la que desea llamar. Esto es, se llama al *trampolín*.

La primera vez que se llama a la entrada de la PLT para una función, se realiza la acción por omisión: llamar al enlazador dinámico para que resuelva el símbolo y *parchee* la entrada de la GOT. Después se llama a la función destino correspondiente. En sucesivas llamadas, la GOT ya tiene *parcheada* la entrada y el trampolín salta directamente a la función destino.

Supongamos que el proceso llama a **`printf` por primera vez** (ver Figura 3.2):

[14] Puedes ver las opciones `now`, `lazy` y `relro` en *ld(1)*.

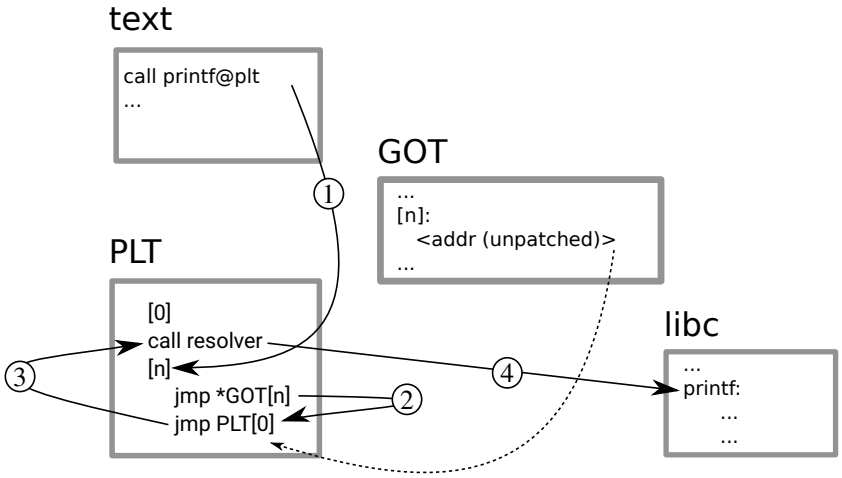

Figura 3.2: Primera llamada a `printf` con *lazy binding*.

1. El texto en realidad llama a la función `printf@plt`.

2. La función `printf@plt` salta a la dirección de la entrada de `printf` en la GOT. Dicha entrada de la GOT está sin parchear, y está inicializada a la dirección siguiente a la instrucción de la PLT que se estaba ejecutando (del propio código de `printf@plt`).

3. Dicha función llama al enlazador dinámico para resolver el símbolo y parchear la entrada de la GOT. Pasa como argumento el índice del símbolo en la tabla.

4. Después de resolver, se salta a la función `printf`.

Las llamadas **posteriores** a `printf` saltarán, en el paso 2, a la función `printf` de la libc, porque la GOT ya está *parcheada*. La Figura 3.3 muestra esa situación.

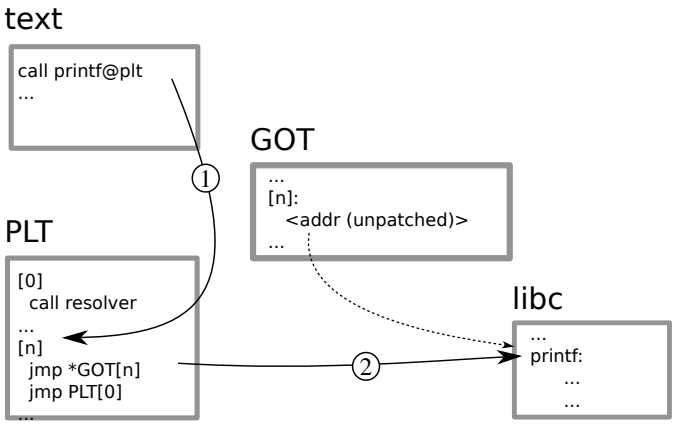

Figura 3.3: Segunda llamada a `printf` con *lazy binding*.

Una biblioteca puede cargarse en la misma dirección de memoria para todos los procesos que la utilizan, pero no siempre es así. Si compilamos con *PIC* (***P**osition **I**ndependent **C**ode*), no ocurrirá de esa forma. Esto se puede querer hacer por razones de seguridad, pero también obstaculiza la depuración. También es común en los sistemas actuales.

Podemos usar el comando `ldd` para ver de qué bibliotecas dinámicas depende un binario ejecutable. Por ejemplo:

```
$ ldd ./hello
        linux-vdso.so.1 (0x00007fff88ffb000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9d358ab000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f9d35e9e000)
$
```

El programa que hemos usado de ejemplo sólo usa la libc y `linux-vdso`, que se utiliza para realizar llamadas al sistema de una forma especial. La última línea que muestra es la del propio enlazador dinámico.

¿De dónde se saca la biblioteca? Bueno, en un sistema GNU/Linux tenemos distintas rutas con bibliotecas. El programa puede buscar en distintos sitios, esto depende del sistema y su configuración. Este es un ejemplo real (por orden de prioridad, si se encuentra se deja de mirar):

1. Directorios indicados en la variable de entorno `LD_PRELOAD`.
2. Si el símbolo contiene barras ("/"), se carga de esa ruta (relativa o absoluta).
3. Directorios indicados en la sección `DT_RPATH` del binario o en el atributo `DT_RUNPATH` del ELF.
4. Directorios indicados en la variable de entorno `LD_LIBRARY_PATH`.
5. En el directorio `/lib`.
6. En el directorio `/usr/lib`.

Si no puede encontrar la biblioteca, al final habrá un fallo de ejecución.

3.4. Propiedades de un proceso

El núcleo del sistema mantiene toda la información sobre los procesos en lo que se denomina **tabla de procesos**, que no es más que una estructura de datos (por ejemplo, una lista enlazada o un array de *records*) cuyas entradas representan los procesos. Estos *records* tendrán distintos campos con la información necesaria para representar a los procesos. Entre otros, tendrá:

- El PID, un número que identifica el proceso. Cada vez que se crea un proceso, se le asigna un id único que lo identifica durante toda su vida.

- El PPID, que es el PID de su creador, también llamado *proceso padre*.
- El estado del proceso: ejecutando, listo para ejecutar, bloqueado, etc.
- La prioridad del proceso, que hará que ejecute más tiempo o menos.
- Los registros de la CPU (para los cambios de contexto).
- El directorio de trabajo actual del proceso.
- El UID y el GID, que son las credenciales del proceso que indican el usuario y el grupo del proceso.
- Los segmentos de memoria (text, data, bss) y el mapa de regiones (que estudiaremos en el tema dedicado a la memoria).
- Los descriptores de fichero, que indican qué ficheros tiene abiertos el proceso.

En el kernel de Linux, esta estructura se llama `struct task_struct`[15].

El comando `ps` nos permite inspeccionar la tabla de procesos. Para ver todos los procesos que están ejecutando, se suelen usar los modificadores `aux` (como siempre, consulta la página de manual para obtener los detalles de estas opciones):

```
 $ ps aux | tail -10
esoriano 28923  0.0  0.0  58164  9600 pts/1    S+   19:33   0:00 vim
root     28945  0.0  0.0      0     0 ?        I    19:35   0:00 [kwo
root     28965  0.0  0.0      0     0 ?        I    19:37   0:00 [kwo
root     29036  0.0  0.0      0     0 ?        I    19:39   0:00 [kwo
esoriano 29084  0.0  0.0  25344  5580 pts/3    Ss   19:40   0:00 bash
root     29324  0.0  0.0      0     0 ?        I    19:47   0:00 [kwo
root     29543  0.0  0.0      0     0 ?        I    19:51   0:00 [kwo
root     29600  0.0  0.0      0     0 ?        I    19:55   0:00 [kwo
esoriano 29617  0.0  0.0  39888  3504 pts/3    R+   19:56   0:00 ps
esoriano 29618  0.0  0.0  10028   768 pts/3    S+   19:56   0:00 tail
 $
```

El comando `ps` tiene muchos modificadores para imprimir distintas propiedades de los procesos. En ese ejemplo, la primera columna es el usuario a nombre del que ejecuta el proceso (el nombre de usuario correspondiente al UID), la segunda es el PID del proceso y la última es el programa que está ejecutando.

También hemos visto ya las variables de entorno de un proceso. En Linux, y en general en Unix, se almacenan en área de usuario[16]. El proceso hereda de su creador una **copia** de sus variables de entorno [17].

[15] Puedes encontrar la definición de esta estructura en el fuente del kernel de Linux, en el fichero `include/linux/sched.h`.

[16] Nótese que en algunos sistemas se almacenan en área de kernel.

[17] En el caso de fork, toda la memoria es una copia, en el de `exec`, se copia a la pila como veremos

En realidad, el tercer parámetro de `main` es un array de cadenas con las variables de entorno, terminado en NULL. El compilador de C no dará un error ni una *warning*, aunque se declare main como que recibe 2 o 3 parámetros (o 1 o ninguno). Sencillamente ignorará los que no están presentes.

Por ejemplo, este programa imprime las cadenas del tercer argumento de `main`:

Programa 3.2: printenvp.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int
4  main(int argc, char *argv[], char *envp[])
5  {
6      int i;
7
8      for (i = 0; envp[i] != NULL; i++) {
9          printf("envp[%d]: %s\n", i, envp[i]);
10     }
11     exit(EXIT_SUCCESS);
12 }
```

Si lo ejecutamos (imprimiendo sólo las 10 primeras líneas):

```
 $ ./prinenvp | head -10
envp[0]: LC_MEASUREMENT=es_US.UTF-8
envp[1]: LESSCLOSE=/usr/bin/lesspipe %s %s
envp[2]: LC_PAPER=es_US.UTF-8
envp[3]: LC_MONETARY=es_US.UTF-8
envp[4]: TERMINATOR_UUID=urn:uuid:487ee28a-5cb8-401e-b4f3-af1a5b9eb7dd
envp[5]: XDG_MENU_PREFIX=gnome-
envp[6]: LANG=en_US.UTF-8
envp[7]: GDM_LANG=en_US
envp[8]: DISPLAY=:0
envp[9]: CLASES=/home/esoriano/prof/clases
 $
```

Internamente, la libc mantiene una variable global llamada `environ` que apunta a las variables de entorno. Al crear una imagen nueva del proceso (*exec*), se copian todas las variables de entorno en la base de la pila del proceso. Si se definen nuevas variables de entorno durante la ejecución, se mueven al heap. Por lo general, no se suele usar el tercer parámetro de `main` y es mejor usar estas funciones de biblioteca para manipular las variables de entorno:

- `getenv`: devuelve un puntero al valor de una variable de entorno.

  ```
  char * getenv(const char *name);
  ```

- `setenv`: escribe el valor de una variable de entorno. Si existe, la puede sobrescribir o no (tercer parámetro). Internamente, hace una copia de la cadena.

```
int setenv(const char *name, const char *val, int o);
```

- `unsetenv`: elimina una variable de entorno.

  ```
  int unsetenv(const char *name);
  ```

En Linux también podemos inspeccionar las propiedades de los procesos en `/proc` (ver *proc(5)*). En realidad así es como funciona el comando `ps`, Para verlo, puedes ejecutar `strace ps`.

Proc es un sistema de ficheros sintético (o virtual). Sus ficheros no son datos almacenados en un disco. Sus datos se generan al vuelo según se leen. En este caso, se generan a partir de las propiedades de los procesos (esto es, de la información que se almacena en la tabla de procesos, etc.).

En `/proc` hay un directorio por cada PID que existe actualmente en el sistema (por tanto, por cada proceso que hay). Dentro de cada directorio tenemos ficheros con información para el proceso correspondiente. Algunos ejemplos son:

- `stat`: propiedades del proceso (PID, PPID, estado...).
- `cmline`: línea que se usó para ejecutar.
- `exe`: el fichero con el programa que está ejecutando.
- `fd` y `fdinfo`: información sobre ficheros abiertos.
- `maps`: regiones de memoria del proceso.
- `mem`: memoria del proceso.

Por ejemplo, si queremos ver cómo se ha invocado el comando que ejecuta el proceso con PID 312:

```
 $ cat /proc/312/cmdline
/bin/bash$
```

Como ya se ha comentado, un proceso tiene *credenciales*. Cuando se ejecuta un programa, éste ejecuta a nombre de un usuario y un grupo. El sistema aplica el **control de acceso** a los objetos (por ejemplo, si un usuario puede leer o escribir un fichero) en base a las credenciales del proceso. En Linux, la información sobre usuarios y grupos se almacena en los ficheros `/etc/passwd`, `/etc/shadow` y `/etc/group`.

El UID es un número que identifica a un usuario. El GID identifica a un grupo. Un usuario puede pertenecer a múltiples grupos. El sistema siempre trabaja con números (UID y GID), los nombres asociados (esto es, las cadenas de caracteres que representan los nombres de usuarios y grupos) se usan para mostrar la información a los humanos, porque reconocemos mejor los nombres que los números.

Durante la vida del proceso las credenciales pueden cambiar. Además, se usan otras credenciales derivadas del UID, como el *EUID* (UID efectivo). Esto depende del tipo de

sistema en el que estemos. En Linux, existen varias credenciales de este tipo con distinto fin. Por ejemplo, para controlar el acceso a los ficheros, se usa el *FSUID* (***F**ile **S**ystem **U**ser **ID***). Para más información, lee *credentials(7)*.

En el terminal podemos usar el comando `id` para comprobar el UID y GID:

```
 $ id
uid=6009(pepe) gid=600(profes) groups=600(profes),20(dialout),29(audio),
46(plugdev),108(kvm),201(android)
 $
```

Las siguientes llamadas al sistema sirven para consultar y cambiar algunas propiedades del proceso:

- `getpid`: devuelve el PID del proceso.

  ```
  pid_t getpid(void);
  ```

- `getppid`: devuelve el PID del creador.

  ```
  pid_t getppid(void);
  ```

- `getuid`: devuelve el UID.

  ```
  uid_t getuid(void);
  ```

- `getgid`: devuelve el GID.

  ```
  gid_t getgid(void);
  ```

- `getcwd`: devuelve el directorio de trabajo. Lo copia en el buffer que pasamos. Si buf es NULL, reserva memoria dinámica.

  ```
  char * getcwd(char *buf, size_t size);
  ```

- `chdir`: cambia el directorio de trabajo actual del proceso. Esto afecta, por ejemplo, cuando se le pasa un path relativo a `open`, por ejemplo `open("src/a.c", O_RDWR);`

  ```
  int chdir(const char *path);
  ```

3.4.1. Muerte de un proceso

Cuando se muere un proceso, se actualiza una propiedad imporante: el **estado de salida** (*status*) del proceso. El creador del proceso usará ese valor para saber si el proceso realizó su trabajo o no lo hizo. Es importante notificar esto. Ya se habló de ello en el Capítulo 2. También es importante que el creador del proceso se preocupe de recoger el estado de salida, para poder liberar los recursos asociados. Ya veremos más adelante cómo se se recoge el estado con la llamada al sistema `wait`.

Para terminar el proceso se usa la llamada al sistema `exit`:

```
void exit(int status);
```

El parámetro que se le pasa a `exit` determina el estado del proceso al acabar: éxito (`EXIT_SUCCESS`, valor 0) o fallo (`EXIT_FAILURE`, valor distinto de 0[18]). Aunque la función recibe un `int`, para el estado de salida sólo se utiliza un byte (valores de 0 a 255).

Si tu programa no llama a `exit` directamente, se llamará cuando `main` retorne: la rutina `_start` de la libc llamará a `exit` para acabar, pasando el valor que retornó `main`. Por eso la función `main` devuelve un `int`, para poder llamar a `exit` con él.

Es importante que, cuando suceda un error, el programa no falle de forma silenciosa. Debería escribir un error[19] y salir con estado de error, es decir, el argumento de `exit` debe ser distinto de 0. Tenemos dos funciones muy útiles cuando queremos acabar el proceso imprimiendo un mensaje de error antes. Con estas funciones, se puede imprimir la causa con una cadena de formato similar a las que se usan con *printf(3)*:

- `err`: escribe el error de la última llamada concatenado después de la cadena con formato que se le pasa, y después termina el proceso llamando a `exit` con el estado indicado en su primer parámetro. Para imprimir el mensaje, usará el valor de `errno` (escribe la cadena asociada a dicho error).

  ```
  void err(int eval, const char *fmt, ...);
  ```

- `errx`: escribe un error con la cadena con formato que se le pasa (sin el error descrito por `errno`) y termina el proceso llamando a `exit` con el estado indicado en su primer parámetro. Se debe utilizar esta función cuando la razón para salir no es el fallo de una llamada a biblioteca que actualice `errno`.

  ```
  void errx(int eval, const char *fmt, ...);
  ```

3.5. Creación de procesos

¿Cómo se crea un proceso? Por ahora sólo hemos creado procesos desde la shell. Cada vez que se ejecuta un comando en la shell, se crea un proceso (o varios, p. ej. si es un pipeline).

Existe una llamada al sistema para crear un proceso: *fork(2)*. Desde que se crea el primer proceso de usuario, todos los procesos se crean usando esa llamada. El shell también crea los procesos con `fork`.

[18] En la práctica, algunos comandos tienen varios posibles estados de fallo. Por ejemplo, como vimos en el capítulo de shell2.9, un programa como `grep`, puede tener un fallo real en el que, por ejemplo, no exista el fichero o dé un error de permisos. Otro tipo de error, es que no se encuentre la cadena a buscar. Lo primero se señaliza con 2, lo segundo con 1. Otro ejemplo es el comando `ssh`. Normalmente sale con el estado de éxito del último comando, pero si fue el propio `ssh` el que falló, por ejemplo porque no se pudo establecer la conexión, sale con estado 255.

[19] Por la salida de error, stderr, el descriptor 2, como se verá más adelante

```
int fork(void);
```

Esta llamada crea un proceso nuevo (hijo) que ejecuta el mismo programa que el proceso creador (padre). Esto permite configurar el proceso creado (hijo) antes de ponerle a ejecutar otro programa distinto, si es ese el objetivo (que no lo es siempre). Los procesos tienen relación padre-hijo. Si queremos ejecutar un programa distinto en un proceso, tenemos una llamada diferente: *execv(3)*[20].

La llamada `fork` crea un nuevo proceso hijo, con un nuevo PID. La función no tiene parámetros y retorna un entero. Desde el punto de vista del padre, es sencillo: llama a `fork`, que retorna cuando ha creado el proceso hijo. Lo peculiar es lo siguiente: el proceso creado (el hijo) también retorna de `fork` como si él hubiera llamado a dicha función. El nuevo proceso ha comenzado su ejecución dentro de la llamada a `fork`!

Imagina que haces un clon exacto de ti mismo, tal y como estás ahora (no un clon tuyo recién nacido). Ese clon no ha comenzado a vivir desde el nacimiento; ha comenzado a vivir desde la clonación. Imagina también que el clon creado tiene un tatuaje que no tienes tú. Al terminar el proceso de clonado, el clon podría pensar que él es el original (tú). Tendría que mirar si tiene el tatuaje para ver si es el original o el clon[21]. Algo similar sucede con `fork`.

Ambos procesos creados con `fork` son copias idénticas indistinguibles. La única diferencia es el valor de retorno de la función, que es diferente en el padre y en el hijo. En el padre, retorna el PID del hijo. En el hijo, siempre retorna 0. En caso de error, no se crea el proceso y retorna -1. Esto permite crear una bifurcación en nuestro programa para que el padre y el hijo realicen acciones diferentes.

En Linux, la función `fork` en realidad realiza la llamada al sistema `clone`, que ya veremos en el tema 7. Veamos un ejemplo simple:

[20] Hay un conjunto grande de funciones en la familia de `exec`, que es el nombre original de la llamada al sistema en Unix. En Linux, esas funciones son recubrimientos de la llama al sistema *execvpe(2)*.
[21] La novela *"Gente de Barro"* [20] tiene un argumento con una idea parecida.

Programa 3.3: fork0.c

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <err.h>
5  #include <stdlib.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     switch (fork()) {
11     case -1:
12         err(EXIT_FAILURE, "fork failed!");
13     case 0:
14         printf("I am the child\n");
15         break;
16     default:
17         printf("I am the parent\n");
18     }
19     exit(EXIT_SUCCESS);
20 }
```

Este programa crea un proceso hijo. El proceso padre entra en el *default* e imprime el mensaje "I am the parent". **Concurrentemente**, el proceso hijo entrará en el caso 0 e imprimirá "I am the child". Ambos procesos terminan su ejecución llamando a *exit(2)*:

```
$ ./fork0
I am the parent
I am the child
$
```

Vemos que el padre ha llegado a escribir el mensaje antes que el hijo. No podemos suponer que es así. Los dos ejecutan de forma concurrente y se pueden ordenar de cualquier forma. Ejecutemos varias veces más el programa:

```
 $ ./fork0
I am the parent
I am the child
 $ ./fork0
I am the parent
I am the child
 $ ./fork0
I am the parent
I am the child
 $ ./fork0
I am the parent
I am the child
 $ ./fork0
I am the parent
I am the child
 $
```

Vaya. ¿No es verdad lo que se ha dicho? Veamos. El siguiente script ejecuta repetidamente el programa mirando si el hijo imprime el mensaje antes que el padre. En ese caso, imprime el número de la ejecución. Cuando ejecutamos ese script, vemos que en ocasiones sí gana el hijo (llega a imprimir antes su línea que el padre). Esto pasa con una probabilidad baja, pero sucede:

```
 $ cat fork0-test.sh
#!/bin/sh

n=0
while true
do
        if ./fork0 | sed 1q | grep -q child
        then
                echo execution $n: child wins
        fi
        n=$((n+1))
done
 $ ./fork0-test.sh
execution 89: child wins
execution 103: child wins
execution 194: child wins
execution 416: child wins
execution 433: child wins
execution 810: child wins
execution 891: child wins
execution 1032: child wins
execution 1634: child wins
execution 1942: child wins
execution 1952: child wins
 ^C
 $
```

Por ese motivo **no se puede suponer** ningún orden. Los dos procesos son concurrentes y si queremos un orden específico (p. ej. que siempre imprima el hijo antes que el padre), necesitamos usar algún mecanismo de sincronización.

Cuando el resultado final depende de la carrera entre los dos procesos tenemos lo que se denomina una *condición de carrera*. En este caso, los procesos comparten otros recursos del sistema (p. ej. ficheros), por lo que las carreras pueden provocar estados incoherentes. Siempre debemos evitar las condiciones de carrera.

El hijo es un clon exacto del padre y es **independiente** del padre: en eso consiste la abstracción *proceso*. Se *copia* toda la memoria del proceso padre al proceso hijo: *text*, *data*, la pila, etc. ¿Qué valor tienen las variables? En el momento de creación, la misma. Luego cada uno modificará su copia de las variables y podrán diferir. Veamos otro ejemplo:

Programa 3.4: fork1.c

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <err.h>
5  #include <stdlib.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     int x = 13;
11
12     switch (fork()) {
13     case -1:
14         err(EXIT_FAILURE, "fork failed!");
15     case 0:
16         x++;
17         printf("I am the child, x is %d\n", x);
18         break;
19     default:
20         x++;
21         printf("I am the parent, x is %d\n", x);
22     }
23     printf("bye bye! x is %d\n", x);
24     exit(EXIT_SUCCESS);
25 }
```

Si ejecutamos el programa, vemos que cada proceso incrementa su copia de la variable x:

```
 $ ./fork1
I am the parent, x is 14
bye bye! x is 14
I am the child, x is 14
bye bye! x is 14
 $
```

Un proceso puede tener muchos hijos, tantos como le permita el núcleo del sistema (en base a los recursos disponibles, que son finitos).

3.5.1. Ejecución de programas

La función *execv(3)* ejecuta un programa en el proceso actual. El parámetro `path` es la ruta del fichero ejecutable (binario o interpretado, dependiendo del *número mágico*) que queremos ejecutar. `argv[]` es un array de strings con los argumentos para el programa, que tiene que terminar en un NULL. El primer elemento del array es el nombre del programa.

```
int execv(const char *path, char *const argv[]);
```

El nuevo programa empezará a ejecutar desde su punto de entrada y su programa principal tendrá como `argv` el array que se le ha pasado a `execv`.

Así se ejecutan todos los programas desde que se crea el primer proceso de usuario (*init*). Cuando ejecutamos un nuevo comando en la shell, se crea un proceso con `fork` y después se ejecuta el comando en el proceso creado con `execv`. Las dos llamadas juntas son muy potentes.

Esto no es fácil de entender. Nuestro programa llama a esa función y, si todo va bien, nunca más retorna: se convierte en otra cosa[22]. Toda la memoria del proceso se inicializa para ejecutar el nuevo programa: se carga el segmento *text*, también *data*, se inicializa el BSS, etc. Si la llamada a `execv` retorna, es que ha tenido un error y no ha podido ejecutar el programa indicado. Eso puede pasar por distintos motivos, por ejemplo que no exista el fichero, que no tengamos permisos para ejecutarlo, etc.

Veamos un ejemplo:

Programa 3.5: execv.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      char *args[3];
9
10     args[0] = "myls";
11     args[1] = "/tmp";
12     args[2] = NULL;
13
14     fprintf(stderr, "hello!\n");
15     execv("/bin/ls", args);
16     fprintf(stderr, "bye!\n");
17     exit(EXIT_FAILURE);
18 }
```

Si ejecutamos, vemos el primer mensaje que se escribe por la salida, y después vemos la salida de ejecutar un `ls /tmp`. No llegamos a ver el mensaje de despedida "bye!". Si retornase, es que ha habido un error en la ejecución.

[22] Como si estuviese poseído, pero permanente.

```
 $ ls /tmp
file1
file2
 $ ./execv
hello!
file1
file2
 $
```

Hay varias versiones de `exec`. Por ejemplo, *execl(3)* es similar, pero se pasan los argumentos de otra forma[23] (uno por parámetro). Esta versión es ideal para cuando se saben los argumentos de antemano. En ese caso, no necesitamos construir un array de strings con los argumentos. El último argumento tiene que ser siempre NULL.

```
int execl(const char *path, const char *arg, ...);
```

El mismo programa anterior se podría escribir así (es más cómodo):

Programa 3.6: execl.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <err.h>
5
6  int
7  main(int argc, char *argv[])
8  {
9
10     fprintf(stderr, "hello!\n");
11     execl("/bin/ls", "myls", "/tmp", NULL);
12     fprintf(stderr, "bye!\n");
13     err(EXIT_FAILURE, "exec failed");
14 }
```

Hay otras funciones de la familia `exec` que inspeccionan la variable de entorno `PATH` para buscar la ruta del ejecutable que se quiere ejecutar (p. ej. *execvp(2)*).

No hay que confundir `execv` y compañía con la función *system(3)*, que lo que hace es ejecutar una shell en un proceso nuevo.

Un detalle importante para que todo funcione es que, aunque `exec` cambie la imagen que ejecuta el proceso, hay una cosa que se hereda y permanece invariante: el *entorno*. Como hemos mencionado antes, la llamada al sistema `exec` copiará en la pila del proceso, junto con el contenido de `argv`, una copia idéntica del entorno que tenía el proceso antes

[23] Se dice que esta es una función variádica (*variadic*): recibe un número de parámetros indeterminado (en este caso, mayor o igual que 2). Un caso similar es `printf`. En C esto se representa con tres puntos en el lugar done iría el último argumento.

de llamar a `exec`. Además, actualizará la variable global `environ` y pasará como tercer argumento de main (que, como se ha comentado anteriormente, normalmente se ignora) una copia de ese puntero.

3.5.2. Esperando por los hijos

Cuando un proceso crea un hijo, puede esperar a que se muera para ver su estado de salida. Ese estado dará información sobre su ejecución: si ha tenido éxito o no.

La función *wait(3)* retorna cada vez que un hijo cambia de estado. Recuerda que un proceso puede crear muchos procesos hijo. Se considera un cambio de estado:

- Que un hijo muera.
- Que un hijo se quede parado (por una *señal*).
- Que un hijo parado continúe ejecutando (de nuevo, por una *señal*).

Ya veremos las señales en el Capítulo 6. Ahora mismo nos interesa sólo la muerte de un hijo. Hay que tener claro que esta llamada no notifica el cambio de estado de los procesos nietos, tataranietos y demás descendientes; sólo notifica el cambio de estado de los hijos directos. Retorna -1 si no hay hijos por los que esperar o en caso de error. En otro caso, retorna el PID del hijo que ha tenido el cambio de estado. Si hay varios hijos con cambio de estado, retorna el de uno de ellos. ¿El de qué hijo? No podemos suponer nada: tenemos que mirar el valor de retorno. Si no hay cambios de estado pendientes pero hay hijos vivos que los pueden generar, al llamar a `wait` el proceso padre se quedará bloqueado hasta que suceda un cambio de estado.

```
pid_t wait(int *wstatus);
```

El parámetro que se le pasa es un entero (pasado por referencia). Dentro de ese entero nos dejará la información sobre el proceso que ha muerto: los 8 bits de menos peso del entero es el estado de salida, el resto de bits se usan para otras cosas (p. ej. para decir si la función `wait` ha retornado por la muerte del hijo o por otro motivo).

Hay que usar las macros descritas en la página de manual para discriminar entre los estados y conseguir el status de salida del hijo:

- `WIFEXITED(status)`: verdadero si el proceso ha terminado llamando a `exit()`.
- `WEXITSTATUS(status)`: evalúa a los 8 bits que describen el estado de salida del proceso (valor que el hijo pasó a `exit()`).
- `WIFSIGNALED(status)`: verdadero si el proceso ha terminado por una señal.
- `WIFSTOPPED(status)`: verdadero si el proceso ha sido parado.

- `WIFCONTINUED(status)`: verdadero si el proceso estaba parado y ahora continuará ejecutando.

En el siguiente ejemplo esperamos por los cinco procesos hijos que se crean. Cada proceso hijo va a dormir durante 10 segundos, ejecutando el comando `sleep`. Ten en cuenta que esta no es la forma correcta de hacer que un proceso duerma un tiempo dado (existe una llamada al sistema `sleep` para eso), es simplemente un ejemplo para ver cómo funciona la espera:

Programa 3.7: wait.c

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <err.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7
8  enum {
9      Nchildren = 5,
10 };
11
12 int
13 main(int argc, char *argv[])
14 {
15     int pid;
16     int sts;
17     int i;
18
19     for (i = 0; i < Nchildren; i++) {
20         pid = fork();
21         switch (pid) {
22         case -1:
23             err(EXIT_FAILURE, "fork failed!");
24         case 0:
25             execl("/bin/sleep", "sleep", "10", NULL);
26             err(EXIT_FAILURE, "exec failed");
27         default:
28             printf("child created: %d\n", pid);
29         }
30     }
31
32     while ((pid = wait(&sts)) != -1) {
33         printf("Did process %d exit?\n", pid);
34         if (WIFEXITED(sts)) {
35             printf("Yes! the status was: %d\n", WEXITSTATUS(sts));
36         } else {
37             printf("No\n");
38         }
39     }
40     exit(EXIT_SUCCESS);
41 }
```

Cuando ejecutamos el programa, podemos ver que el padre escribe los cinco mensajes avisando de que ha creado un nuevo proceso y durante 10 segundos no se imprime nada. Después, veremos que la llamada a `wait` retorna una vez por cada proceso hijo. Morirán

todos *"a la vez"*,[24], no en un orden concreto, a los 10 segundos. Por cada uno, el padre podrá escribir el estado de salida del hijo (o imprimir que no si ha pasado algo distinto). El estado será 0 porque el comando `sleep` saldrá con éxito:

```
 $ ./wait
child created: 10779
child created: 10780
child created: 10781
child created: 10782
child created: 10783
Did process 10779 exit?
Yes! the status was: 0
Did process 10781 exit?
Yes! the status was: 0
Did process 10780 exit?
Yes! the status was: 0
Did process 10782 exit?
Yes! the status was: 0
Did process 10783 exit?
Yes! the status was: 0
 $
```

Cuando un proceso muere, no se puede eliminar su estructura de la tabla de procesos hasta que el padre no recoja su estado. Un proceso muerto pendiente de que se recoja su estado (*status*) es un *zombie*. Hay que tener cuidado: un proceso *zombie* sigue ocupando su posición en la tabla de procesos, por lo que sigue consumiendo recursos.

Si el padre muere antes que los hijos, estos pasan a ser huérfanos (*orphan*). Alguien se tiene que hacer cargo de esperar por los huérfanos. Por eso, en Unix los huérfanos pasan a ser hijos del proceso `init` (el proceso con PID 1). El proceso `init` llama a `wait` por todos sus hijos para evitar que se queden *zombies* para siempre.

Esto era así originalmente en Unix. Desde hace tiempo, en Linux y otros sistemas modernos, se puede modificar este comportamiento[25] y hacer que algunos procesos pasen a ser hijos de un proceso que no sea `systemd`[26].

[24] Así es como lo percibirá el usuario. Esos procesos concurrentes morirán todos aproximadamente 10 segundos después de haber ejecutado el comando `sleep`.

[25] El nombre que recibe el proceso que hereda la adopción de procesos huérfanos descendientes de un proceso dado es *subreaper*. Hereda la adopción de todos los descendientes de ese proceso si se quedan huérfanos y no tienen un *subreaper* más cercano en el árbol genealógico. Mediante la llamada `prctl` se puede cambiar el *subreaper* de un proceso. Para ese proceso hace, además, que el *subreaper* pase a ser, efectivamente, su padre. Ver `prctl(2)`.

[26] `systemd` es el *init* de la mayoría de los sistemas Linux actuales.

3.5.3. Job control

En la shell, cuando usamos el `&` para ejecutar en background, lo que indicamos es que no se debe hacer `wait` por el proceso creado antes de mostrar el *prompt* y volver a leer otro comando. Esto tiene que ver con el *job control* del terminal.

Cada comando (simple o pipeline) que está ejecutando es un trabajo (*job*). No hay que confundir un *job* con un proceso: un *job* puede contener o referirse a uno o varios procesos. El comando `jobs` lista los trabajos actuales en esta shell.

La combinación de teclas `Ctrl+c` (pulsar a la vez la tecla Ctrl y la tecla c) interrumpe el trabajo que está en primer plano. En este caso, por omisión, mueren todos los procesos del trabajo. La combinación de teclas `Ctrl+z` para el trabajo que está en primer plano.

El comando `bg` reanuda, en segundo plano, un trabajo parado. El comando `fg` trae a primer plano un trabajo.

Veamos un ejemplo. Primero miro qué procesos se están ejecutando en el terminal, listo los trabajos (no hay ninguno), ejecuto un `sleep` de 1000 segundos en primer plano, lo paro[27] con un `Ctrl+z`, lo mando a ejecutar en segundo plano, ejecuto otro `sleep` de 50000 segundos en segundo plano, listo los trabajos (están los dos `sleep`), traigo a primer plano el trabajo 2 (el `sleep` de 50000 segundos), lo interrumpo (y muere) con `Ctrl+c` y por último listo los procesos que se están ejecutando en el terminal:

[27] En el ejemplo, la interrupción y la parada con el teclado aparece en color rojo.

```
 $ ps
  PID TTY          TIME CMD
13058 pts/5    00:00:00 ps
25863 pts/5    00:00:00 bash
 $ jobs
 $ sleep 1000
^Z
[1]+  Stopped                 sleep 1000
 $ bg
[1]+ sleep 1000 &
 $ sleep 50000 &
[2] 13060
 $ jobs
[1]-  Running                 sleep 1000 &
[2]+  Running                 sleep 50000 &
 $ fg 2
sleep 50000
^C
 $ ps
  PID TTY          TIME CMD
13059 pts/5    00:00:00 sleep
13134 pts/5    00:00:00 ps
25863 pts/5    00:00:00 bash
 $
```

En el Capítulo 6 volveremos a hablar del *job control*.

3.6. Planificación

El núcleo del sistema se encarga de *repartir* las CPUs entre todos los procesos que quieren ejecutar. Esto es lo que hace un componente del kernel que se llama **planificador** (*scheduler*). En esta sección estudiaremos el funcionamiento del planificador.

Para estudiar la planificación, podemos simplificar y resumir los posibles estados de un proceso en:

- Ejecutando: el proceso está ahora mismo ejecutando en una CPU, los registros de un core de la CPU contendrán estado del proceso, el PC apuntará a texto asociado al proceso, etc.
- Listo para ejecutar: el proceso no está ejecutando pero está listo para ejecutar. Está en la cola de *listos para ejecutar*. En cuanto el planificador quiera, lo pondrá a ejecutar en una CPU (retirando al que estaba ejecutando hasta ahora).
- Bloqueado: el proceso no está listo para ejecutar, está esperando a que suceda un evento. Por ejemplo, el proceso está esperando a que termine una operación de

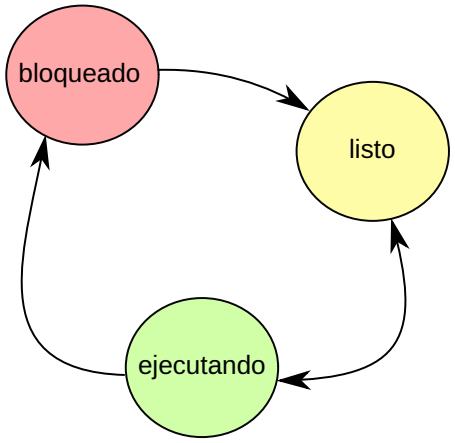

Figura 3.4: Los posibles estados simplificados de un proceso y sus transiciones.

entrada/salida para poder continuar. En este caso, el planificador no puede elegirlo para ejecutar en una CPU.

La Figura 3.4 muestra las posibles transiciones entre estados. Un proceso ejecutando puede bloquearse esperando por un evento. También puede dejar de ejecutar porque el planificador decida que ya no le toca más tiempo de CPU, pasando a estar listo para volver a ejecutar. Un proceso bloqueado, cuando el evento por el que esperaba sucede, pasa a estar listo para ejecutar.

Cuando el planificador decide que le toca ejecutar a un proceso que está listo para ejecutar, se realiza un *cambio de contexto* y el proceso pasa a estar ejecutando.

Otra vez más, debemos separar políticas de mecanismos. El mecanismo de la planificación es el *cambio de contexto*. Grosso modo, consiste en:

1. Salvar el estado de los registros del procesador en la estructura de datos que representa al proceso saliente en el kernel.

2. Cargar el estado de los registros del proceso entrante en el procesador.

Tenemos que tener en cuenta que realizar un cambio de contexto es costoso y tiene implicaciones que entenderemos más adelante cuando veamos la gestión de memoria en el Capítulo 5.

La *política de planificación* es la que dicta cómo se reparten las CPUs entre los procesos que están listos para ejecutar.

Para simplificar, supondremos a partir de ahora un sistema con una única CPU[28]. También veremos los procesos como ráfagas de operaciones de CPU y operaciones de entrada/salida: un proceso está ejecutando instrucciones hasta que se bloquea para hacer

[28]El modelo de programación (concurrencia) hace indistinguible que haya una CPU o varias. Lo que cambia es cuantos procesos pueden estar realmente ejecutando código a la vez (paralelismo si son varios, pseudoparalelismo si es uno).

entrada/salida; cuando acaba la entrada/salida, pasa a estar listo para ejecutar. Esta simplificación nos ayudará a comprender mejor el trabajo del planificador.

Algunos procesos están dominados por CPU y otros procesos están dominados por entrada/salida. Los primeros tienen ráfagas largas de CPU y hacen poca entrada/salida. Los segundos, al contrario, dedican poco tiempo a procesar y tienen mucha entrada/salida.

No podemos tener una política que satisfaga todos estos requisitos [2, Capítulo 4]:

- Justicia (*fairness*): todos los procesos tienen que tener su tiempo de CPU, hay que evitar la **hambruna** (inanición, *starvation*). La hambruna se da cuando hay procesos que nunca pueden ejecutar porque no se les da tiempo de CPU.
- Eficiencia (*efficiency*): sacar máximo partido a la CPU, minimizar el tiempo en el que la CPU está ociosa (esto es, el tiempo en el que la CPU está sin hacer trabajo útil).
- Tiempo de respuesta interactiva: cuando el usuario está esperando respuesta, es importante el tiempo de espera en la cola hasta que se empieza con el proceso que le atiende. Por ejemplo, al mover el ratón el cursor puede tardar tiempo en mover el puntero en la pantalla. Esa latencia puede llegar a hacer inutilizables ciertas aplicaciones interactivas.
- Tiempo de respuesta (*turnaround*): tiempo de entrega del resultado final de un cómputo.
- Rendimiento (*throughput*): cantidad de trabajo acabado por unidad de tiempo.

La planificación puede ser:

- No expulsiva (*non-preemptive*, colaborativa): el proceso abandona la CPU porque ha terminado su ráfaga de CPU y se bloquea haciendo E/S o por decisión propia llamando a una operación para abandonar la CPU, que se suele llamar *yield* [29]. La única diferencia entre estos dos casos es que en el primero se bloqueará y en el segundo pasará a estar listo para ejecutar inmediatamente.
- Expulsiva (*preemptive*): el planificador puede expulsar a los procesos cuando lo decida, aunque no hayan terminado su ráfaga de CPU. Cuando un proceso está ejecutando, ocurren interrupciones de reloj que hacen que el control pase al núcleo del sistema para que éste pueda efectuar tareas periódicas de gestión. Una de esas tareas puede ser ver si el proceso que está ejecutando actualmente ya lleva demasiado tiempo ejecutando y le toca a otro. Si decide que lleva demasiado tiempo, el planificador realiza un cambio de contexto (aunque el proceso que estaba ejecutando no haya acabado aún su ráfaga de CPU) porque *se ha acabado su turno*.

[29] En Unix una llamada a *sleep(3)* de 0 segundos es equivalente a un *yield*.

3.6.1. Políticas

A continuación describiremos algunas de las políticas básicas, principalmente para ilustrar los problemas que hay que solucionar en la planificación. Este asunto se trata en profundidad en las referencias [2, 1].

La primera que veremos es la política no expulsiva más sencilla: *FCFS* (***F**irst **C**ome **F**irst **S**erve*). Los procesos listos para ejecutar se ponen en una cola (FIFO). Cuando un proceso que está ejecutando termina su ráfaga de CPU y necesita bloquearse para hacer entrada/salida, el planificador pone a ejecutar el primer proceso que esté en la cola. Cuando un proceso termina su entrada/salida, se pone al final de la cola. Esta política tiene un tiempo promedio de espera alto. Además, provoca lo que se llama el efecto *convoy*. Esto ocurre cuando un proceso limitado por CPU (A) está mucho tiempo ejecutando sus instrucciones, impidiendo que otros procesos limitados por entrada/salida (B y C) ejecuten sus pequeñas ráfagas y se pongan a hacer entrada/salida de nuevo. Como se retrasa el momento de iniciar las operaciones de entrada/salida de B y C, se malgasta capacidad de entrada/salida (y tiempo para terminar las tres tareas en conjunto). Este efecto se puede contrarrestar dando prioridad a los trabajos más cortos.

Otra política no expulsiva llamada *SJF* (***S**hortest **J**ob **F**irst*) consiste en dar prioridad a los procesos con ráfaga más corta. Para ello, el planificador tiene que conocer el volumen del trabajo de antemano. Esta política mejora el tiempo de espera promedio pero introduce un nuevo problema: la **hambruna**. Imaginemos un proceso A al que le toca una ráfaga de CPU de tamaño L, pero siempre hay procesos listos para ejecutar con una ráfaga de CPU menor que L. En ese caso, A nunca puede ejecutar; estará listo para ejecutar para siempre.

Esta política puede evolucionar a una política expulsiva llamada *SRTF* (***S**hortest **R**emaining **T**ime **F**irst*). Cuando entra en la cola de procesos listos para ejecutar un proceso (A) con una ráfaga de CPU más corta que lo que le queda al proceso actual (B) por ejecutar, se hace un cambio de contexto de A a B. Esta política tiene el mismo problema: tiene hambruna. Imagina que siempre llegan procesos con ráfagas más cortas que las de un proceso que está listo para ejecutar desde hace tiempo. En ese caso, dicho proceso nunca va a poder ejecutar.

La política expulsiva *Round-Robin* (R-R) consiste en tener una cola FIFO, pero un proceso sólo puede ejecutar un tiempo determinado antes de que sea expulsado de la CPU[30]. Ese tiempo se llama **cuanto**. Si la ráfaga de CPU es más larga que el cuanto, el proceso agotará su cuanto y tendrá que seguir más tarde cuando le vuelva a tocar (será puesto al final de la cola). Si el proceso acaba su ráfaga de CPU durante su cuanto, se bloqueará para hacer entrada/salida y el planificador pondrá a ejecutar el siguiente proceso listo para ejecutar (el primer proceso de la cola). En realidad, es más fácil ver esta política como una ruleta (de ahí su nombre). Esta política aumenta la respuesta interactiva pero reduce el *throughput* (la cantidad total de trabajo útil realizado por unidad de tiempo), ya que se provocan más cambios de contexto.

El problema fundamental de esta política es elegir el tiempo del cuanto. Si el cuanto es pequeño, se desperdicia mucha CPU en los cambios de contexto y baja el *throughput*.

[30] Lo que comúnmente llamaríamos por turnos y en algunos ámbitos se llama multiplexación en el tiempo

Si es cuanto es grande, las aplicaciones interactivas sufren. Hay que tener en cuenta que si el cuanto es enorme, al final esta política degenera en FCFS, ya que ningún proceso sería expulsado por agotar su cuanto.

Otras políticas aplican prioridades en base a la *importancia* de los procesos. Por ejemplo, si un reproductor de video no llega a procesar a tiempo el siguiente *frame* del video, veremos que se corta el video. Sin embargo, si un cliente de correo se retrasa un segundo para comprobar si tenemos correos por leer, no notaremos nada como usuarios interactivos. Con este fin, a los procesos se les asignan distintas **prioridades**.

Otro ejemplo puede ser un proceso que controla el actuador de una aeronave vs. un proceso que computa estadísticas de vuelo. En ese caso el proceso que controla el actuador tiene requisitos críticos muy duros: si no se ejecutan sus instrucciones a tiempo hay problemas serios. Este tipo de aplicaciones necesitan tener plazos estrictos garantizados para ejecutar sus instrucciones. A eso se le llama **planificación de tiempo real** (*Real Time*). No todos los sistemas tienen un planificador de este tipo.

En general, la prioridad es *estática* si nunca cambia o *dinámica* si puede cambiar mientras ejecuta el proceso. Normalmente, es dinámica.

Siempre que tengamos prioridades, se corre el riesgo de tener hambruna. Una estrategia para evitarlo es el envejecimiento (*aging*): tener en cuenta la edad del proceso, de tal forma que según va envejeciendo un proceso, su prioridad va subiendo automáticamente.

En Unix, se denomina *niceness* a la prioridad de un proceso desde el punto de vista de usuario (para que la pueda ajustar). Un proceso puede ser más o menos *majete* (*nice*) dependiendo de si cede o no cede su tiempo de CPU al resto. En Linux, el valor va del valor -20 (máxima prioridad, el menos majo) al valor 19 (mínima prioridad, el más majo).

Un proceso puede ajustar su nivel con la llamada al sistema *nice(2)*. La llamada *getpriority(2)* sirve para consultar el valor. En general, la prioridad se hereda del proceso padre (el estándar POSIX lo deja a elección de la implementación) y se conserva después de un `exec`. Dependiendo del sistema, el proceso puede necesitar privilegios para ajustar su valor.

Internamente, el núcleo maneja sus niveles de prioridad para el planificador. Por ejemplo, Linux maneja un nivel de prioridad de planificación llamado *pri*. Cuando ajustamos el valor de *nice*, cambia ese valor.

El comando `nice` sirve para ejecutar un programa indicando el nivel de prioridad. El comando `renice` sirve para cambiar la prioridad de un proceso.

Veamos un ejemplo: lanzamos un proceso para ejecutar el comando `sleep` con un `nice` de 15, lo paramos y lo mandamos a ejecutar en segundo plano. Podemos ver el valor de *nice* en la columna `NI` y el valor de *pri* en la columna `PRI`:

```
 $ nice --adjustment=15 sleep 1000
^Z
[1]+  Stopped                 nice --adjustment=15 sleep 1000
 $ bg
[1]+ nice ---adjustment=15 sleep 1000 &
 $ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000 22891 14826  0  80   0 -  6343 wait   pts/3    00:00:00 bash
0 S  1000 25217     1  0  80   0 - 379560 poll_s pts/3   00:00:19 evince
0 S  1000 29174 22891  0  95  15 -  2497 restar pts/3    00:00:00 sleep
4 R  1000 29175 22891  0  80   0 -  7858 -      pts/3    00:00:00 ps
 $
```

En realidad, un planificador de un sistema moderno es más complicado que lo que hemos visto. Si quieres detalles sobre el planificador de Linux, que está basado en Round-Robin y tiene capacidades de tiempo real, puedes leer la página de manual *sched(7)*.

Un planificador de un sistema moderno puede aplicar distintas políticas y tener múltiples colas para procesos de distintas prioridad (planificador multinivel), cada cola puede aplicar un algoritmo distinto, distintos métodos para mover de una cola a otra, etc. [2, Capítulo 4]. Por ejemplo, un sistema con planificación multinivel con retroalimentación [1, Sección 2.5.3]:

- Tiene múltiples colas para distintas prioridades.
- Se aplica la política Round-Robin en cada cola.
- Si hay procesos listos en una cola, no se atienden las colas de menor prioridad.
- Aplica prioridades dinámicas: cuando un proceso agota su cuanto, se baja su prioridad. Cuando un proceso no agota su cuanto, se sube su prioridad.

El hardware también es más complicado y eso afecta a la planificación. Hoy en día, tenemos máquinas multiprocesador con distintas arquitecturas. El planificador tiene que ser consciente de la arquitectura:

- Las arquitecturas *UMA* (***U**niform **M**emory **A**ccess*) son aquellas en las que todas las CPUs son iguales y el tiempo de acceso a las diferentes regiones de la memoria física es similar. En estos sistemas se aplica una planificación *SMP* (***S**ymmetric **M**ulti-**P**rocessor*): cualquier proceso puede ser planificado en cualquier CPU y puede haber una cola de planificación global.

  Aún así, en estas arquitecturas el planificador debe tener en cuenta las cachés de memoria de las CPUs. Si un proceso ejecutó por última vez en la CPU 3, es mejor que vuelva a ejecutar en la CPU 3 porque la caché de ese procesador estará poblada con datos de ese proceso. Si se pone a ejecutar en la CPU 5 estaremos desperdiciando esa caché. En este caso se dice que el planificador tiene que tener en cuenta la *afinidad* de los procesos con los procesadores.

- En las arquitecturas *NUMA* (***N**on-**U**niform **M**emory **A**ccess*) algunas partes de la memoria y ciertos dispositivos están *más cerca* de unas CPUs que de otras. En estas arquitecturas, una CPU está conectada únicamente a algunos chips de memoria. Si necesita acceder a otros chips, es necesario que se comunique con la CPU que está conectada a esos chips. Para ello tiene que usar una red que interconecta los procesadores (tejido de coherencia), por lo que es mucho menos eficiente que acceder a la memoria a la que está conectada directamente. Esto también ocurre para acceder a los dispositivos.

  En este caso, el planificador tiene que tener muy en cuenta dónde ha ejecutado un proceso anteriormente, ya que su memoria estará en las partes de memoria física más cercana a ese procesador (si la gestión de memoria es eficiente, claro). Ignorar la afinidad tiene un impacto tremendo en el rendimiento del sistema. En estos sistemas se aplica una *afinidad dura*: los procesos sólo deben planificarse en ciertas CPUs.

- Otras arquitecturas multiprocesador son heterogéneas, mezclan CPUs de distintos tipos. Por ejemplo unas CPUs son menos rápidas que otras o tienen instrucciones más apropiadas para ciertos trabajos, etc. En este caso, el planificador tiene que hacer su trabajo teniendo en cuenta los detalles del proceso que tiene que planificar. Hoy en días hay CPUs que comparten partes del procesador (por ejemplo *Hyperthreading*), unidades (por ejemplo de coma flotante) que pueden encenderse o apagarse por razones de consumo, etc. Este es un campo en continuo desarrollo.

3.6.2. Depurando programas

Aunque no está relacionado estrictamente con este capítulo, ya entendemos lo mínimo de enlazado para entender las cosas que nos vamos a encontrar al ejecutar `gdb` para depurar nuestros programas[31]. El comando `gdb` es un depurador que nos permite:

- Inspeccionar un programa desensamblándolo, es decir, mostrando sus instrucciones de código máquina.

- Inspeccionar un proceso. Esto incluye ver valores de la memoria, como variables locales o globales, ver por dónde va ejecutando en un determinado momento, etc.

- Inspeccionar un *fichero de core*:[32] la *foto* de un proceso en un fichero. En el fichero *core* está el contenido de todos los registros y de la memoria del proceso en un punto de ejecución. Lo puedes ver como el cuerpo muerto del proceso que usamos para hacer una autopsia.

[31] Como vimos en el tema de `make` 2.29.2, en algunos proyectos GNU, `info` es la mejor manera de documentarse sobre un comando. Te recomendamos que instales `gdb-doc`, `apt install gdb-doc` y ejecutes `info gdb` .

[32] En este caso, el término *core* hace referencia a un volcado de la memoria de un proceso (*core dump*) en un fichero que usamos para depurar, no a una unidad de procesamiento (el *core* de un procesador). El término viene del tipo de memoria RAM que usaban los ordenadores antiguos (hasta la década de 1970), que estaba construida con núcleos (*cores*) magnéticos.

En la mayoría de las ocasiones, usar el depurador **no es la forma más eficiente** de depurar un programa. Esto es importante. Hay que tener en cuenta que el depurador es maquinaria pesada. Depurar un proceso introduce interferencia en su ejecución, requiere entender y tener presentes muchos detalles innecesarios para resolver el problema (esto es, lo que estoy depurando). La mayor parte de las veces (errores lógicos, fallos sencillos de programación, etc.), es mejor poner trazas en el código. Si te duele la garganta y vas al médico, no entras al quirófano directamente. Primero, tu médico te ausculta y te mira la garganta con una linterna. Por lo habitual, no será necesario pasar a quirófano.

Sin embargo, hay ocasiones en las que un depurador es fundamental. Por ejemplo, cuando no tenemos el código fuente del programa que falla o el problema es intratable y estamos intentando orientarnos para ver qué sucede.

También es muy interesante usar un depurador cuando un programa tiene un fallo de ejecución (por ejemplo un *segmentation fault*), sólo para saber cuando sucedió y qué acceso a memoria lo ha provocado (el depurador nos podrá dar información sobre qué línea de tu código ha provocado el fallo).

Para poder depurar con toda la información, tienes que usar la opción `-g` de `gcc` para que el ELF contenga símbolos de depuración.

Ejemplos de uso básico

- Para ver en qué estado está un programa que muere por atravesar un puntero incorrecto (*segmentation fault*):
  1. Arrancamos `gdb` con el ejecutable: `gdb ejecutable`.
  2. Ejecutamos dentro de `gdb` (por ejemplo, proporcionando dos argumentos al programa) con `run argumento1 argumento2`.
  3. Ahora se produce el fallo en ejecución.
  4. Con `bt` se vuelca la pila.
  5. Con `frame 3` selecciono el registro de activación que deseo inspeccionar.
  6. Con `info locals` veo el valor de las variables locales.
  7. Con `info args` veo el valor de los argumentos
  8. Con `whatis z` veo el tipo de la variable `z`.
- Para inspeccionar las funciones de un programa mediante desensamblado:
  1. Arrancamos `gdb` con el ejecutable: `gdb ejecutable`.
  2. Con `disass mifuncion` se obtiene el desensamblado de dicha función:

```
Dump of assembler code for function mifuncion:
   0x000000000000131b <+0>:     endbr64
   0x000000000000131f <+4>:     push   %rbp
   0x0000000000001320 <+5>:     mov    %rsp,%rbp
   0x0000000000001323 <+8>:     sub    $0x40,%rsp
```

```
0x0000000000001327 <+12>:    mov    %edi,-0x34(%rbp)
0x000000000000132a <+15>:    mov    %esi,-0x38(%rbp)
```

- Para meter un punto de ruptura y ejecutar paso a paso:

  1. Arrancamos `gdb` con el ejecutable: `gdb ejecutable`.
  2. Con `break f1` metemos un punto de ruptura en la función `f1`.
  3. Con `run argumento1 argumento2` empezamos la ejecución (por ejemplo, pasando dos argumentos).
  4. La ejecución se para cuando el flujo llega al punto de ruptura de la función `f1`. Ahora podemos inspeccionar como en el ejemplo anterior. Podemos usar `i r` para ver el contenido de los registros, podemos volcar la pila como hemos visto antes, etc.
  5. Con `stepi` ejecutamos una única instrucción. Podemos repetir esto cuantas veces queramos para ver cómo evoluciona la ejecución.
  6. Con `continue` ordenamos que el programa siga ejecutando normalmente.

Estos son sólo tres ejemplos sencillos. El depurador `gdb` es muy potente y permite hacer muchas otras cosas. Hay libros completos sobre el uso de esta herramienta [21].

3.7. Ejercicios no resueltos

1. Utiliza `ldd` para obtener información sobre el `cat` que hay instalado en tu sistema. ¿Qué biblioteca de C se usa? †∗

2. Utiliza `ltrace` y `strace` para saber qué hace realmente `cat`, pruébalo con diferentes argumentos. †∗

3. Utiliza `readelf` para obtener información sobre el binario de `cat` que hay instalado en tu sistema: qué secciones tiene, para qué máquina está compilado... †∗

4. Descubre qué variables de entorno hay en tu sistema viendo la salida de `env` y averigua para qué sirve cada una. †∗

5. Ejecuta `top` y `htop` e intenta ver qué procesos hay ejecutando en tu máquina, para qué sirven y de donde vienen. Recuerda que `dpkg -S` te dice de qué paquete software viene un fichero. †∗

6. Escribe un comando `true` y `false` que hagan lo mismo que los comandos del sistema con ese nombre. Escribe un sólo comando que haga lo que hacen ambos, comportándose de forma diferente dependiendo de cómo le llamen. †∗

7. Escribe un programa en C que tenga un hijo que imprima su PID y duerma (con *sleep(3)*) mucho tiempo. Mira cómo lo representa el programa `pstree`. Redirige sus entradas y salidas a ficheros con `dup`. Mira su directorio en `/proc` y en particular sus descriptores de fichero `/proc/PID/fd`. †∗

8. Escribe en C el comienzo de una minishell. Es un programa que recibe como argumentos un comando y sus argumentos, ejecuta dicho comando y deja en la variable de entorno `RESULT` el estado de salida del comando. Se puede suponer que los comandos están en `/bin`. † † ∗∗

9. Escribe un programa en C que esté en un bucle infinito escribiendo el carácter 'z' por su salida. Ejecútalo con la salida en `/dev/null`. Mira la carga del sistema (por ejemplo con `top`. Cambia su prioridad con `renice` y observa qué le pasa a la carga del sistema. †∗

10. Escribe dos programas, uno en C y otro un script de shell que ambos actúen como una *fork bomb*. Una *fork bomb* es un programa que sólo hace `fork` (en un bucle). Como consecuencia, sus hijos hacen `fork` y se crean un número exponencial de procesos. Atención: no lo ejecutes sin haber salvado todos los ficheros antes. †∗

11. Pon un reproductor de video como `vlc` a reproducir una película. Utiliza `renice` para ver qué efectos tiene cambiar la prioridad de sus procesos sobre el vídeo y el audio. Observa la carga del sistema (por ejemplo con `top`). Ejecuta simultáneamente otros procesos, por ejemplo `yes > /dev/null` para cambiar la carga del sistema y ver qué ocurre. Manipula también su prioridad. †∗

12. Compila y enlaza con `-g` un programa que llame a `write`. Intenta desensamblar la función `write` con `gdb`. Enlaza con `-static`. Desensambla `write`. Observa las diferencias con el caso anterior. ¿Qué instrucción utiliza para entrar en el kernel? † † ∗

4 Ficheros

4.1. ¿Qué es un fichero?

En este capítulo estudiaremos los sistemas de ficheros y el almacenamiento secundario (discos). Ya conocemos la abstracción de fichero, llevamos tiempo usándolos: son datos con un nombre asociado. Normalmente asociamos intuitivamente los ficheros con el almacenamiento secundario: es una forma de tener datos persistentes, es decir, de que no se borren cuando se apaga el ordenador. Están pensados para permitir un gran tamaño, acceso concurrente y proporcionar un sistema de control de acceso para su protección.

No obstante, la abstracción de fichero va más allá: es una **interfaz** para acceder a objetos con ciertas operaciones básicas: comenzar a usar el fichero (i.e. abrir, por ejemplo, para comprobar existencia, pedir permisos y reservar los recursos necesarios), leer, escribir y terminar de usarlo (i.e. cerrar, por ejemplo, para liberar los recursos).

De hecho, hay ficheros que no sirven para tener datos persistentes como hemos explicado en el párrafo anterior, sino para representar dispositivos o acceder a información generada *al vuelo* (por ejemplo para inspeccionar el sistema, como los ficheros que están en `/proc` en un sistema Linux). A ese tipo de ficheros se les llama *ficheros virtuales* o *ficheros sintéticos*. Por ejemplo, puedes usar un fichero sintético para encender y apagar un LED: cuando escribes la cadena "on" en el fichero se enciende y cuando escribes "off" se apaga, sin almacenar nada en almacenamiento secundario. Otro ejemplo, en Linux podemos hacer esto para saber cuánto tiempo lleva el sistema arrancado:

```
 $ cat /proc/uptime
16001.03 122010.87
 $
```

En este caso, no se ha leído ningún dato que esté en almacenamiento secundario. Al leer de ese fichero, el núcleo ha generado dicha información a partir de sus estructuras de datos internas.

Los ficheros pueden estar implementados de distinta forma en distintos sistemas operativos. En algunos sistemas los ficheros pueden ofrecer en el API de lectura un tipo de datos (un fichero de enteros, de estructuras, etc.). En Unix, un fichero es simplemente una secuencia de bytes con un nombre asociado (y algunos otros metadatos como su tamaño, permisos, etc.). Interpretar esos datos es labor de la aplicación, el sistema operativo es *agnóstico*. En Windows un fichero tiene nombre y extensión, en Unix no. En Mac OS X los ficheros pueden tener dos partes, una para los datos y otra llamada *resource fork* (usada para metadatos, icono, etc.). Como se puede ver, cada sistema implementa sus ficheros de una forma un poco distinta.

Se llama **sistema de ficheros** al componente del sistema operativo que implementa los ficheros. Normalmente un sistema de ficheros forma parte del núcleo, aunque también se pueden tener sistemas de ficheros implementados en área de usuario. Aunque es un poco confuso, se llama también **sistema de ficheros** a las estructuras de datos que se almacenan en el disco (y que se inicializan cuando se formatea). Diferentes sistemas de ficheros utilizarán diferentes estructuras de datos.

En las siguientes secciones veremos cómo se implementan los sistemas de ficheros para guardar datos en almacenamiento secundario.

4.2. Discos

Internamente, los discos están divididos en pequeñas partes llamadas sectores o bloques. El bloque (o sector) *físico* es la unidad mínima que permite el soporte hardware leer o escribir. Tienen un tamaño pequeño, normalmente 512 bytes, pero hay discos que tienen sectores físicos de 4096 bytes. Adicionalmente, algunos discos tienen una capa de abstracción software interna (firmware) que permite manejar los sectores como si fueran de otro tamaño, usando *bloques lógicos*. Por ejemplo, si el software sólo puede manejar sectores de 4096 bytes, un disco con sectores físicos de 512 bytes puede configurarse para recibir en el bus bloques lógicos de 4096 bytes. Cuando se lee o se escribe el disco, siempre se hace en base al tamaño de bloque lógico y el disco traduce internamente (lógico a físico). Para evitar confusión, nos referiremos a los sectores del disco (físicos o lógicos) como *bloques de disco*.[1]

El sistema de ficheros también maneja el concepto de bloque (lógico). Esto también puede resultar confuso. Normalmente, un sistema de ficheros gestiona los datos de los ficheros con agrupaciones de tamaño fijo de bloques de disco por cuestiones de eficiencia. En algunos sistemas (p. ej. Windows/MS-DOS), esas agrupaciones se denominan *clusters*. En otros sistemas se refieren a ellos como bloques (lógicos). A partir de ahora nos referiremos a ellos como *bloques del sistema de ficheros*.

Por tanto, un *bloque del sistema de ficheros* (unidad mínima de datos dentro del sistema de ficheros) es una agrupación de *bloques de disco* (unidad mínima de datos en la interfaz hardware del disco).

Como en cualquier caso en el que se discretice en paquetes el reparto de un recurso asociado a datos (trozos de memoria, trozos de disco etc.) tendremos problemas de *fragmentación*. Si ejecuto esto en una shell:

```
 $ echo hey > afile
 $ ls -l afile
-rw-rw-r-- 1 esoriano esoriano 4 abr 28 16:53 afile
 $
```

¿Cuántos bytes de almacenamiento está consumiendo ese fichero? Podemos ver que el fichero tiene 4 bytes ("hey\n"). ¿Ocupa sólo 4 bytes del disco? No.

Como poco consume un bloque de disco. Si los bloques de disco son de 512 bytes, como poco se usarán 512 bytes para guardar esos datos. Eso significa que se malgastan 508 bytes de almacenamiento secundario para almacenar ese fichero.

[1] La denominación es confusa. Tanto la literatura como los fabricantes son poco coherentes en el uso de sector o bloque. Además, es difícil saber qué es físico y qué lógico dados los diferentes niveles de abstracción en el propio hardware. Por ello y para dejar claro a qué nos referimos, cuando haya dudas, añadiremos el adjetivo lógico o físico según corresponda e intentaremos dejar claro a qué nos referimos.

Pero es peor todavía. ¿Cuántos bloques de disco consume cada bloque del sistema de ficheros? Veamos:

```
 $ sudo blockdev --getbsz /dev/sda5
4096
 $
```

El bloque del sistema de ficheros en este caso es de 4096 bytes, 8 bloques de disco. El fichero tiene un tamaño de 4 bytes pero en realidad gasta 4096 bytes. Por tanto, se malgastan 4092 bytes. A esto se le llama fragmentación. Este tipo de fragmentación se denomina *fragmentación interna*.

Cuanto más grande sea el bloque del sistema de ficheros, más fragmentación interna habrá, pero más efectiva será la entrada/salida (porque leer bloques contiguos suele ser más rápido que leer bloques dispersos) y más ligeras serán las estructuras de datos internas del sistema de ficheros (tendrán que mantener menos referencias a bloques). Hay que llegar a un compromiso. Por lo general, se usan bloques del sistema de ficheros de 4096 bytes.

El comando `stat` nos da más información sobre el fichero que hemos creado antes. Todo cuadra, el fichero consume 8 bloques de disco aunque su tamaño sea de 4 bytes:

```
 $  stat afile
  File: afile
  Size: 4               Blocks: 8          IO Block: 4096 regular file
Device: 805h/2053d      Inode: 2368344     Links: 1
Access: (0664/-rw-rw-r--)  Uid: (1000/esoriano)   Gid: (1000/esoriano)
Access: 2020-04-28 16:53:38.897157509 +0200
Modify: 2020-04-28 16:53:48.536939279 +0200
Change: 2020-04-28 16:53:48.536939279 +0200
 Birth: -
 $
```

Pero además, el sistema tiene que dedicar más de 4096 bytes para mantener ese fichero. No estamos teniendo en cuenta los bloques que se tienen que usar para guardar los *metadatos* del fichero. Los metadatos son los datos sobre los datos, como el tamaño del fichero, sus permisos, su fecha de modificación, etc. Vemos que el comando `stat` nos ofrece algunos de los metadatos del fichero que le pasamos como argumento.

Por estas razones (entre otras), si sumamos el tamaño de los ficheros veremos que no cuadra con el espacio de disco usado/libre. No es que te hayan timado con el disco, es que los ficheros ocupan más de lo que parece (o menos, como veremos más adelante).

Los medios de almacenamiento se pueden clasificar por su forma de acceso. Si un medio de almacenamiento proporciona *acceso secuencial* es el que hay que recorrerlo bloque a bloque. Por ejemplo, para leer un dato de las cintas que se usaban hace mucho tenías que avanzar o retroceder hasta ese punto (rebobinar). Ojo, todavía hoy se usan cintas para sistemas de backup, pero no es común. Por ejemplo, Sony presentó en 2017 pequeñas

cintas con una capacidad de hasta 330 terabytes y en 2020 IBM fabricaba dispositivos de cinta de hasta 350 petabytes.

Si un medio de almacenamiento proporciona *acceso aleatorio*, entonces puede acceder a un dato directamente sin tener que recorrer el medio.

Los discos duros mecánicos tradicionales están formados por discos magnéticos apilados que giran y unas cabezas que leen/escriben su superficie. Por esa razón, los discos más antiguos usan direccionamiento CHS (***C**ylinder, **H**ead, **S**ector*) para acceder a los bloques físicos. En ese caso, la propia dirección (el número que identifica un bloque en el disco) tiene relación con la geometría interna del disco. Desde hace tiempo los discos usan direccionamiento LBA (***L**ogical **B**lock **A**ddressing*), que es independiente de la geometría interna del disco[2]. Con LBA, cada bloque de disco tiene asignado una dirección lineal (un número). El firmware de los discos modernos es mucho más complejo que el de los discos antiguos y gestiona la traducción a la geometría interna.

Hoy en día también es habitual usar discos SSD, (***S**olid-**s**tate **D**rive*), que están hechos con un chip (típicamente memoria Flash). En general, son más rápidos que los discos magnéticos, pero también tienen inconvenientes (son más caros, duran menos tiempo, soportan menos escrituras, etc.). También existen discos híbridos, SSHD (***S**olid-**s**tate **H**ybrid **D**rive*) que mezclan ambas tecnologías. Adicionalmente, muchos discos tienen pequeños sistemas embebidos con RAM y cachés internas, firmware que detecta cuando un bloque está dañado o se ha utilizado mucho y lo marca para no usarlo, etc.

4.2.1. Dispositivos y particiones

Un disco se puede dividir en partes llamadas **particiones**. El software trata cada partición como un disco separado. Podemos tener un sistema de ficheros distinto en cada partición.

En Unix, los dispositivos se representan con ficheros en `/dev`. Esto tiene una ventaja: para manipular los dispositivos podemos usar las mismas herramientas que usamos para manipular los ficheros convencionales. Por ejemplo, el comando `dd` es muy útil para copiar datos entre dispositivos y podemos ver cuántas particiones tiene un disco usando el comando `ls`.

Los dispositivos de `/dev/` siguen un esquema de nombrado. En Linux, los discos duros comienzan por `sd` seguido de la posición en el bus (a, b, etc.), seguido del número de partición. Por ejemplo, `/dev/sda2` representa la segunda partición del primer disco del bus SATA y `/dev/sdb1` representa a la primera partición del segundo disco del bus SATA.[3]

Hay dos tipos de dispositivos:

[2] Pero todavía podemos ver direcciones CHS en algunos sitios por compatibilidad hacia atrás.

[3] El nombre concreto del dispositivo y el esquema de nombrado ha ido cambiando dependiendo de la tecnología y por tanto, del driver concreto que lo sirve. En el momento de escribir esto, en Linux, ha aparecido recientemente un driver NVMe para discos de estado sólido, SSD. Aquí la notación sería `/dev/nvme0n1p1` para el disco 0, primer espacio de nombres, primera partición (el equivalente a `/dev/sda1`).

- Un dispositivo de **caracteres** trabaja con flujos de bytes, podemos tratar la información byte a byte, como un teclado.
- Un dispositivo de **bloques** trabaja con trozos de datos de cierta longitud (bloque). El dispositivo tiene un tamaño determinado y se tiene acceso aleatorio a sus bloques.

Los discos son dispositivos de bloques. Eso es lo que indica la primera letra (b) de la salida de `ls -l`:

```
 $ ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 abr 28 11:59 /dev/sda
brw-rw---- 1 root disk 8, 1 abr 28 11:59 /dev/sda1
brw-rw---- 1 root disk 8, 2 abr 28 11:59 /dev/sda2
brw-rw---- 1 root disk 8, 3 abr 28 11:59 /dev/sda3
brw-rw---- 1 root disk 8, 4 abr 28 11:59 /dev/sda4
brw-rw---- 1 root disk 8, 5 abr 28 11:59 /dev/sda5
brw-rw---- 1 root disk 8, 6 abr 28 11:59 /dev/sda6
 $
```

Cada dispositivo tiene asociados dos números que sirven al núcleo para identificar el dispositivo:

- *Major number*: indica la clase del dispositivo (i.e. su driver), todos los de la misma clase tienen el mismo número mayor.
- *Minor number*: indica la instancia concreta dentro de esa clase de dispositivo.

Como vemos en el ejemplo anterior, todos los ficheros listados (las particiones del primer disco) tienen 8 como *major* y cada uno tiene un *minor* diferente.

Las particiones de un disco se definen en su *tabla de particiones*. El firmware del sistema impone el formato que tiene que tener la tabla de particiones y dónde se encuentra el cargador del sistema operativo. La tabla de particiones no es más que un conjunto de bloques donde se anota qué particiones hay. El sistema operativo podría mirar en otro sitio y tener unas particiones alternativas, pero recuerda que el firmware (UEFI[4]) también tiene que leer eso (para poder buscar el kernel y arrancar). Son sencillamente datos en el disco, pero también es un convenio compartido entre el firmware, el cargador y el sistema operativo.

En los PCs viejos con firmware BIOS la tabla de particiones se guardaba en el primer bloque del disco, llamado MBR (***M**aster **B**oot **R**ecord*). En 512 bytes tenía que entrar todo esto:

[4]Como veremos, es UEFI o BIOS en PCs de arquitectura Intel/AMD. En otras arquitecturas es diferente pero similar. Por ejemplo en placas con procesadores ARM hay un firmware que se llama U-Boot. En los teléfonos Android es el Android Boot Loader (también llamado Little Kernel o LK).

- Cargador primario: 440 bytes.
- Tabla de particiones primarias. Sólo se pueden tener 4 particiones, cada una de las entradas es de 16 bytes para indicar:
  - Si es arrancable.
  - Dirección del primer bloque de la partición (CHS).
  - Tipo de partición (tipo de sistema de ficheros).
  - Dirección del último bloque (CHS).
  - Dirección del primer bloque (LBA).
  - Número de bloques de la partición.

Este esquema usa direcciones de 32 bits, por lo que tiene limitaciones. Por ejemplo, no se pueden usar discos de más de 2 TB.

Los PCs modernos tienen un firmware que se llama UEFI. En UEFI se sigue un esquema llamado GPT (***G****UID* ***P****artition* ***T****able*). En este esquema sólo se usan direcciones LBA y tamaños de 64 bits (eliminando las restricciones que existían en el esquema viejo). El primer bloque de disco (LBA 0) se reserva para tener compatibilidad hacia atrás con el esquema antiguo. El esquema entero es como sigue:

- Primer bloque (LBA 0), *legacy MBR*. Se reserva por compatibilidad hacia atrás con BIOS.
- Segundo bloque (LBA 1), *GPT header*, con referencias a la siguientes partes.
- A partir de LBA 2 (LBA 2 al X), *GPT Table*, que es un array de bloques con la tabla de particiones. El array tiene un tamaño mínimo de 16 Kb independientemente del tamaño del sector físico. Las entradas en la tabla (128 bytes) contienen el tipo de partición, la dirección LBA del primer bloque, la del último bloque, los atributos (*read-only*, oculta, etc.), nombre de partición, etc.
- A continuación, está el primer bloque usable de datos (aquí pueden empezar las particiones).
- Después está el cargador, en una partición de tipo ESP (***EFI*** ***S****ystem* ***P****artition*[5]).
- Luego (LBA, de sz-2-X al sz-2) tenemos una copia de la tabla de particiones (*GPT Table*, copia del LBA 2 al X).
- El último bloque (LBA sz-1) tiene una copia de la cabecera (*GPT Header*, que está en la dirección LBA 1).

[5]ESP es en realidad un sistema de ficheros de tipo FAT como los que veremos más adelante.

El array *GPT Table* tiene un tamaño mínimo de 16 KB, independientemente del tamaño del bloque del disco. Por tanto, podemos tener tantas particiones como queramos (están limitadas pero el número es muy grande): es mucho más flexible que el sistema anterior.

En Linux, el comando `fdisk` (ver *fdisk(8)*) permite manipular las particiones de un disco[6]. Por ejemplo, podemos ver la tabla de particiones del disco representado por `/dev/sda`:

```
 $ sudo fdisk -l /dev/sda
Disk /dev/sda: 894.3 GiB, 960197124096 bytes, 1875385008 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 6BD56F5D-C7B1-4F61-9EFB-3ED345135C00

Device          Start        End   Sectors   Size Type
/dev/sda1          40     409639    409600   200M EFI System
/dev/sda2      409640  880040943 879631304 419.5G Apple Core storage
/dev/sda3   880040944  881310479   1269536 619.9M Apple boot
/dev/sda4   881311744  913309695  31997952  15.3G Linux swap
/dev/sda5   913309696 1313310719 400001024 190.8G Linux filesystem
/dev/sda6  1313310720 1875384319 562073600   268G Linux filesystem
 $
```

Vemos que la primera partición es de tipo EFI (ESP). También vemos que el esquema de particiones es GPT. Las dos siguientes particiones tienen sistemas de ficheros de sistemas Apple (como HFS+). Las tres siguientes particiones son de Linux (partición de intercambio y dos con sistemas de ficheros EXT4).

También hay herramientas gráficas más intuitivas para manipular las particiones de un disco, como `gparted`. Hay que tener mucho cuidado a la hora de manipular las particiones, ya que podemos perder datos si nos equivocamos.

4.2.2. Virtualización de almacenamiento

Un sistema de ficheros puede conseguir los bloques de discos reales o de otros dispositivos virtuales.

Por ejemplo, con **RAID** (***R**edundant **A**rray of **I**nexpensibe **D**isks*) podemos crear discos virtuales a partir de discos físicos. Esto se puede implementar en hardware (una tarjeta que instalamos en la placa madre) o en software (en un driver del sistema operativo). Hay muchos tipos de RAID, algunos son:

[6] Ojo, también puede borrar y sobrescribir la tabla de particiones y provocar que se pierda información del disco.

- **RAID 0** permite crear un disco virtual a partir de varios discos de igual tamaño. Lo que se hace es intercalar sus bloques (lo que se denomina *interleaving* o *stripping*) consiguiendo un disco virtual cuyo tamaño es la suma del tamaño de los discos que lo forman. Por ejemplo, con dos discos, el bloque 0 es el bloque 0 del disco A, el bloque 1 es el bloque 0 del disco B, el bloque 2 es el bloque 1 del disco A, etc.

  La ventaja es que los bloques contiguos se escriben en paralelo (porque van a distintos discos). Por tanto es más rápido. El inconveniente es que el fallo de uno de los discos estropea el disco virtual completo.

- **RAID 1** implementa un espejo (*mirror*). Los discos que lo forman son copias exactas. Cada escritura de un bloque de disco se tiene que realizar en todos los discos (se puede hacer en paralelo). Una lectura se puede hacer de cualquiera de los discos.

  La ventaja es que hay redundancia, y si falla uno de los discos, no perdemos los datos. Además permite leer distintos bloques en paralelo (uno desde cada disco del RAID).

- **RAID 5** también hace *striping* (como RAID 0) para tener un disco virtual grande a partir de discos más pequeños. La diferencia es que en este caso se mantiene redundancia. La idea es usar varios discos iguales, dedicando el tamaño de un disco completo para redundancia (paridad) con el fin de soportar el fallo de uno de los discos.

  Los datos y la paridad están distribuidos entre los distintos discos (no se dedica un disco para sólo para mantener redundancia). Con este tipo de RAID, si falla uno de los discos, se puede reconstruir en un disco nuevo con los datos de paridad. Si fallan dos, se pierde el disco virtual. A la hora de escribir un bloque de datos, debemos escribir en dos discos (el dato y la paridad).

- **RAID 1+0** mezcla de RAID 0 (*striping*) y RAID 1 (*mirroring*). Por ejemplo, usando 4 discos: se crean dos RAID 1 (espejos) y con esos dos espejos se crea un RAID 0. Este tipo de RAID busca un compromiso entre redundancia y rapidez.

Estos son algunos tipos de RAID, hay más. Puedes encontrar más información sobre RAID en [22] y [23].

Hay otras formas de virtualización de almacenamiento. Algunos sistemas operativos pueden configurar **volúmenes lógicos** (LV), que son como particiones virtuales compuestas por distintas particiones reales de distinto tamaño (que además pueden venir de distintos discos). Esto nos permite tener volúmenes muy grandes o ampliar en demanda el tamaño de un volumen si nos quedamos sin espacio. En Linux, esto se llama LVM.

Las máquinas virtuales usan **imágenes de disco**, que no es otra cosa que usar un fichero como si fuera un disco. Ese fichero que tiene dentro la estructura completa (bloque a bloque) de un disco duro, disco óptico, etc. Hay distintos formatos, como `iso`[7], `bin`,

[7] Técnicamente, iso es un sistema de ficheros, iso9660, para dispositivos ópticos como CDs, bin es una imagen de disco en crudo, dmg es un sistema de ficheros para instaladores de Mac OS X y VDI es un formato de imagen de disco para máquinas virtuales de Virtualbox.

`dmg`, `VDI`, etc. Además de ser útiles para las máquinas virtuales, también lo son para realizar copias de seguridad, clonar sistemas, transmitir discos completos por la red, etc. Más adelante veremos cómo utilizarlas mediante el *dispositivo de loopback*.

Hoy en día también hay sistemas **SAN** (***S**torage **A**rea **N**etwork*) que proporcionan bloques de disco a través de la red. Son, literalmente, un disco duro conectado a la red. Cuando se necesita un bloque de disco, se pide a través de la red (p. ej. ethernet). Hay distintas tecnologías de este tipo, por ejemplo iSCSI, ATA over Ethernet (AoE) o HyperSCSI.

Como se puede ver, un sistema de ficheros puede usar bloques de disco procedentes de distintos tipos de dispositivos físicos o virtuales.

4.2.3. Planificación de entrada/salida

El núcleo del sistema operativo se tiene que encargar de ordenar las operaciones de entrada/salida (leer y escribir bloques del disco) que requieren los procesos que están usando los ficheros. Para ello tiene que usar algún algoritmo de planificación de entrada/salida.

Estos algoritmos necesitan detalles sobre el funcionamiento interno los dispositivos de almacenamiento: hay que saber la distancia que hay entre los bloques de disco. El problema es que hoy en día no tenemos ni idea del tipo de almacenamiento que se usará ni de cómo funcionan por dentro muchos dispositivos.

Si hablamos de discos magnéticos, cuando se usaba direccionamiento CHS era fácil saber cómo de lejos estaban dos bloques físicos (se podía calcular con la dirección de los bloques). Con LBA no tenemos esa información. Además, es difícil saber qué hacen los discos por dentro. Internamente ejecutan software que puede reasignar direcciones a los bloques y tienen memoria caché RAM y de estado sólido.

Linux supone que dos direcciones LBA cercanas también son físicamente cercanas. Esto puede ser cierto, o no. Algunos fabricantes de discos duros aseguran que intentan que esto sea así. Pero, ¿qué pasa si no se trata de un disco magnético? Actualmente tenemos discos SSD (estado sólido) y discos híbridos. En la actualidad, la situación es todavía peor. Puede que los bloques de disco vengan de cualquiera de las soluciones de almacenamiento virtual que se han explicado anteriormente.

Hay distintas políticas de planificación de entrada/salida. Para entenderlas, lo mejor es ver el disco como algo lineal. Cuanto más lejos esté en la línea un bloque, más tiempo tardamos en conseguirlo.

La politica más sencilla es **FIFO**: las operaciones de entrada/salida se meten en una cola por orden de llegada. Es justa y no altera el orden de las operaciones[8], pero resulta lento si operaciones seguidas acceden a bloques lejanos entre sí: hay que estar recorriendo distancias largas continuamente.

La política **Shortest Seek First** da prioridad a las operaciones sobre bloques cercanos a la posición actual. Esto desordena operaciones y además no es justo, por lo que

[8] Esto puede ser importante si se va la luz en mitad de una modificación. Si cambio el orden de las escrituras, puedo terminar con un sistema de ficheros que tiene referencia a datos que no están en el disco.

puede provocar hambruna. Imagina que continuamente llegan operaciones para bloques cercanos. En ese caso, una operación sobre un bloque lejano no se realizará nunca.

El algoritmo del **Ascensor** es un compromiso justicia/eficiencia, no provoca hambruna y desordena operaciones. Consiste en desplazarse sólo en un sentido e ir realizando las operaciones según se recorre la línea, como hace un ascensor que sube o baja (va parando en las plantas que le pillan de paso).

Estos y otros algoritmos de planificación de entrada/salida están descritos en las referencias clásicas de sistemas operativos [2, 1]. Pueden ser de interés si necesitamos crear un sistema de propósito específico que use un hardware de almacenamiento concreto (y conozcamos su funcionamiento interno).

4.3. Implementación de sistemas de ficheros

El problema fundamental que tiene que resolver un sistema de ficheros es asignar el espacio necesario (los bloques de disco) a los ficheros, lo que se anotará en estructuras de datos en el disco. Para ello se pueden seguir distintas estrategias. A continuación describiremos las usadas por algunos de los sistemas de ficheros más populares. Puedes encontrar más información sobre estos aspectos en las referencias bibliográficas clásicas [2, 1].

4.3.1. Asignación contigua

Algunos sistemas de ficheros usan **asignación contigua**. En este caso, un fichero ocupa una serie de bloques contiguos en disco. Tiene ventajas, el acceso es rápido porque es secuencial (todos los bloques del están juntos) y el acceso a un bloque es directo porque sabemos en qué bloque empieza el fichero y cuántos bloques ocupa.

Sin embargo, hay un problema: cuando reservamos el espacio para un fichero, le estamos asignando un tamaño máximo. Detrás de ese fichero habrá otro, por lo que no podrá crecer más.

Además, se crea *fragmentación externa*. Si se van creando y borrando ficheros dinámicamente, al final quedarán huecos entre ellos que, en general, resultarán inservibles. Si sumamos el tamaño de todos esos huecos inservibles, el espacio malgastado puede ser muy grande. Una solución a ese problema es la *compactación* (o *defragmentación*) que consiste en mover todos los ficheros al principio del disco para eliminar los huecos. Esto es muy costoso.

Este tipo de asignación no es apropiada para sistemas de ficheros convencionales en discos duros. Entonces, ¿no se usa este tipo de asignación? Sí, para algunos sistemas de ficheros es deseable. Los discos ópticos (DVD, CD, etc.) suelen ser de sólo lectura y cuando los creamos sabemos el tamaño de todos los ficheros.

4.3.2. Sistemas de ficheros FAT

Como hemos visto, en general es necesario asignar a los ficheros espacio no contiguo (i.e. bloques que no sean contiguos). Para ello, hay que guardar una lista de los bloques

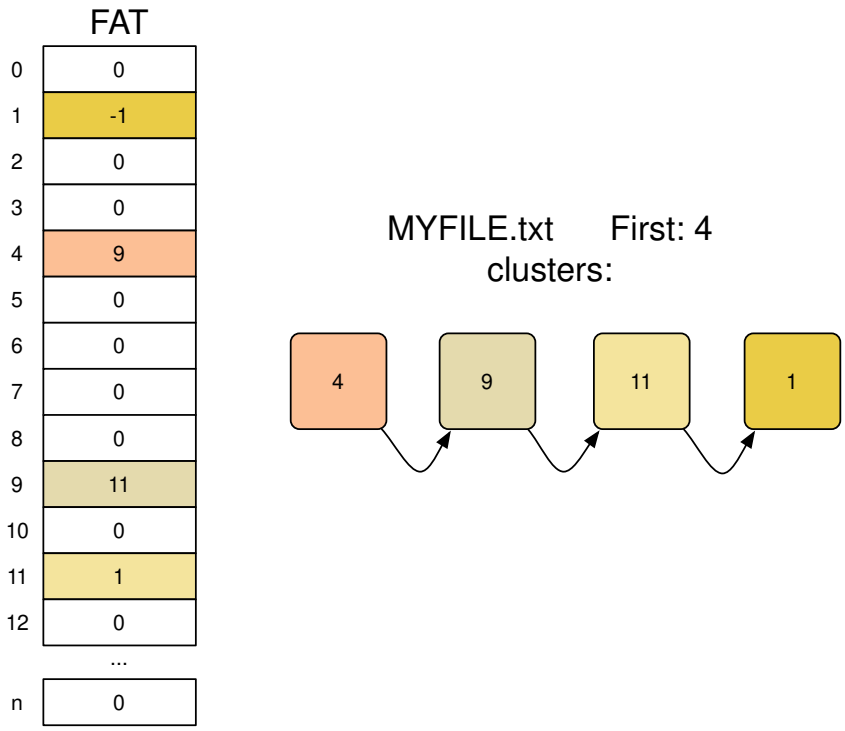

Figura 4.1: Ejemplo de tabla FAT.

que conforman el fichero.

Esto se puede implementar con una lista enlazada, escribiendo la referencia al siguiente bloque en una cabecera dentro del propio bloque. Sin embargo, es más habitual implementarlo como una **lista enlazada con tabla**. En una lista enlazada con tabla, los enlaces de la lista enlazada se mantienen en una tabla separada de los nodos (no se pone dentro del bloque el enlace al siguiente)[9].

En esto se basa la familia de sistemas de ficheros FAT *(**F**ile **A**llocation **T**able*) implementados en los sistemas de Microsoft antiguos (MS-DOS y Windows) y usados actualmente en dispositivos pequeños (cámaras, pen drives, etc.) y en particiones de arranque (por ejemplo en *ESP* de *EFI*).

Las FAT usan bloques del sistema de ficheros muy grandes, que denominan *clusters*. La tabla FAT tiene una entrada por cluster. El índice de la tabla es el número de cluster. Dado un cluster de comienzo, mediante la tabla FAT obtenemos una cadena de clusters de datos. Cada una de estas cadenas representará un fichero o directorio.

En el caso de un fichero, en estos clusters de datos estará el contenido del fichero.

Los clusters de datos de un directorio almacenan una *entrada de directorio* por cada fichero y directorio que contiene (recordemos que, para el usuario, un directorio es un contenedor de ficheros y directorios). La entrada de directorio tiene el nombre, los

[9]De esta forma se separan los datos del sistema de ficheros propiamente dicho (la tabla) de los datos de los ficheros aumentando la localidad y disminuyendo la fragmentación en los metadatos.

Figura 4.2: Partición de tipo FAT32.

metadatos y el número del primer cluster de dicho objeto. A partir de la dirección del primer cluster, se pueden conseguir los siguientes clusters mirando la tabla. En resúmen, un directorio es como un fichero especial, que en sus clusters de datos contiene entradas de directorio una tras otra.

Tanto para un fichero, como para un directorio, para conocer el número del siguiente cluster de datos sólo hay que indexar la tabla FAT con el número del cluster actual.

Veamos el ejemplo ilustrado en la Figura 4.1. Queremos leer el fichero `MYFILE.txt` y el directorio en el que se encuentra dice que su primer cluster es el 4. Después de leer el 4, tendremos que leer el 9 como indica la posición 4 de la tabla. Después el 11 como indica la posición 9 de la tabla. Después el 1, que es el último, ya que la posición 1 tiene el valor -1.

El número del primer cluster del directorio raíz está anotado en el primer sector de la partición. Esto es importante para poder empezar a navegar la FAT al montar el sistema de ficheros (hacerlo visible en el sistema, como veremos más adelante).

Esta forma de asignar el espacio mejora el acceso aleatorio respecto a una lista sin tabla, ya que no hay que leer los clusters del fichero para conseguir la dirección del cluster en una posición dada. Sólo hace falta leer la tabla FAT (que se intentará tener completa en memoria RAM).

Si se corrompe la tabla FAT, perderemos los ficheros (no sus datos, sino su ordenación). Por eso se suele guardar otra copia de la tabla FAT en la partición. Como ya hemos comentado, la tabla no puede ser muy grande porque necesitamos tenerla en memoria. Por eso se usan clusters grandes.

Existen muchas variantes de sistemas de ficheros FAT, en el que cambia principalmente el tamaño de los enteros que se utilizan para direccionar los clusters, FAT12 (12-bit), FAT16 (16-bit), FAT32 (32-bit) y exFAT (64-bit). Algunos, como el último además quita limitaciones de varios tipos como el tamaño máximo de cadena de texto soportada para los nombres.

Veamos como ejemplo FAT32. La Figura 4.2 muestra la estructura interna de una partición FAT32. El bloque 0 de la partición es el *boot sector* (también está duplicado en el bloque 6). Contiene código del cargador secundario e información sobre el sistema de ficheros: número de bloques (son 4 bytes, por tanto tiene un tamaño máximo de 2 TB), etiqueta del volumen, número de copias de la tabla FAT, el primer cluster del directorio raíz, etc. Los clusters de FAT32 son de 32 KB.

Un directorio tiene como datos una lista de entradas de directorios. En FAT32, una entrada de directorio ocupa 32 bytes y contiene:

- Nombre del archivo, 8 bytes.
- Extensión del archivo, 3 bytes.

- Atributos del archivo, 1 byte.
- Campo para uso reservado, 10 bytes.
- Hora de la última modificación, 2 bytes.
- Fecha de la última modificación, 2 bytes.
- Primer cluster del archivo, 4 bytes.
- Tamaño del archivo, 4 bytes (no tiene por qué ocupar todo el último cluster).

Como se puede observar, el tamaño de un fichero (4 bytes) es $2^{32} = 4$ GB. Esa es la razón por la que no se pueden copiar ficheros tan grandes en algunos *pen drives* (sin formatearlos y cambiar su sistema de ficheros).

4.3.3. Sistema de ficheros Unix

Hay otra forma de asignar el espacio. La **asignación indexada** consiste en dedicar bloques únicamente a tener referencias a otros bloques. A un bloque de referencias se le llama *bloque de indirección*. El problema es definir el tamaño de los bloques de indirección: si es grande se desperdicia espacio y si es pequeño, se limita mucho el tamaño de los ficheros. Tiene otro inconveniente: para el acceso a un dato siempre hay que hacer dos operaciones. Primero hay que leer el bloque de indirección para encontrar la dirección del bloque de datos y después hay que leer el bloque de los datos.

Para poder tener ficheros muy grandes, se puede aplicar un esquema multinivel en el que tenemos unos bloques de indirección de primer nivel que tienen las referencias a bloques de indirección de segundo nivel y así sucesivamente (para el número de niveles definido). Por ejemplo, si tenemos dos niveles, los bloques de índice de primer nivel apuntan a bloques de índice de segundo nivel y los bloques de índice de segundo nivel apuntan a bloques de datos. Ahora se agrava el problema de los accesos: para leer un dato se necesitan N accesos a disco.

Lo que hace el sistema de ficheros de Unix es seguir un esquema de **indexado combinado**. Los bloques de datos de un fichero se referencian de distinta forma: algunas referencias directas a bloques de datos, algunos bloques de indirección simple, algunos bloques de indirección doble, etc. De esta forma nos quedamos con lo mejor de cada modelo anterior.

En Unix la estructura que representa a un fichero o directorio se llama *i-nodo*. Es donde residen los metadatos del fichero o directorio y las referencias a los bloques con sus datos. La Figura 4.3 muestra el esquema.

Una partición Unix contiene lo siguiente (ver Figura 4.4):

- Bloque de arranque (*boot block*): cargador secundario.
- *Superbloque*: tamaño del volumen, número de bloques libres, lista de bloques libres, tamaño del vector de i-nodos, siguiente i-nodo libre, cierres...

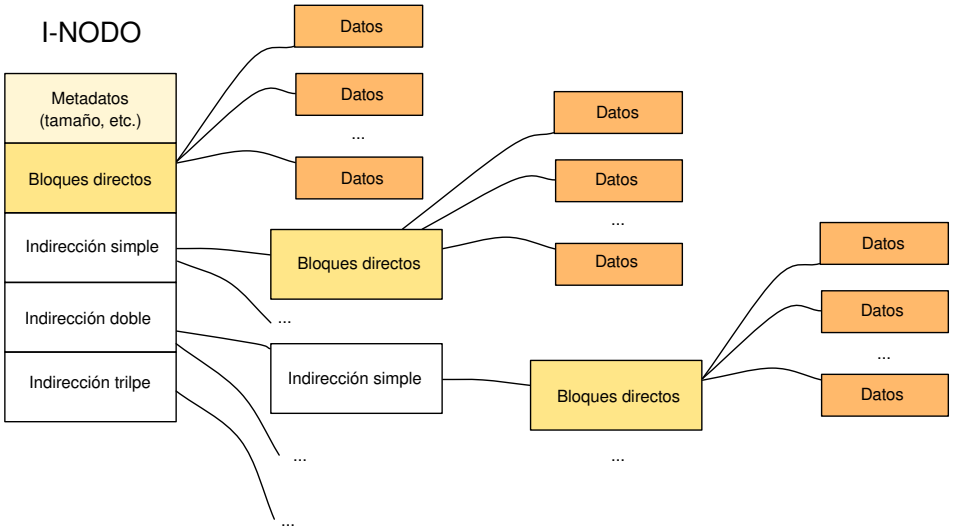

Figura 4.3: I-nodos en Unix.

- Vector de i-nodos: un array de estructuras i-nodo que sirven para almacenar los metadatos de los ficheros y directorios.
- Bloques de datos (datos de los ficheros y directorios).

Cada fichero/directorio tiene una estructura i-nodo asociada, definida por *un número de i-nodo* (que es su índice en el vector de i-nodos). El directorio raíz (/) siempre tiene el número de i-nodo 2. El i-nodo 0 se usa como un valor nulo. Los primeros números de i-nodo están reservados para usos especiales. Por ejemplo, en EXT4 (el sistema de ficheros estándar de Linux), el i-nodo 1 se usa para mantener una lista de los bloques defectuosos, el i-nodo 3 para las cuotas de disco de los usuarios, etc.

El núcleo mantiene una caché de i-nodos en memoria para evitar estar leyendo continuamente el vector de i-nodos. Ese es uno de los motivos por los que hay que *desmontar*[10] un sistema de ficheros de forma ordenada, en la memoria tenemos datos que todavía no se han escrito en su sitio (en el disco).

Un i-nodo contiene, entre otras cosas:

- Los permisos del fichero.
- Las fechas del último acceso a datos (`atime`), la última modificación de los datos (`mtime`) y la última modificación del propio i-nodo (`ctime`).
- Tamaño del fichero. Recuerda, el último bloque de datos seguramente no se use entero, necesitamos saber hasta dónde se usa.
- El usuario que es dueño del fichero y su grupo.

[10] Desmontar significa desconectar el sistema de ficheros, apagarlo. Ya se hablará más adelante de este asunto.

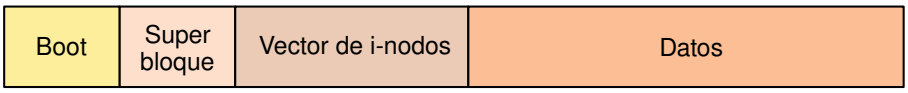

Figura 4.4: Partición Unix.

- El *tipo*[11] de fichero. Ya sabemos que hay ficheros normales y directorios. Ya hemos visto también que hay algunos otros ficheros especiales, como los que representan dispositivos de bloque y de caracteres. Hay otros: enlaces simbólicos, *fifos* y *sockets*.
- Contador de referencias (enlaces o *links*).
- Información sobre los bloques de datos que forman parte del fichero.

¿Te has dado cuenta de que falta una cosa importante del fichero? El i-nodo no contiene el nombre de fichero. Los nombres de los ficheros están en los directorios.

Los directorios tienen su propio i-nodo. Son básicamente ficheros cuyos bloques de datos contienen listas de *entradas de directorio*. Aunque básicamente sean un fichero, el sistema no los trata como tal (p. ej. no se crean o leen con las mismas operaciones, etc.). En un directorio no puedes escribir lo que te de la gana, sólo puedes escribir entradas de directorio con operaciones específicas a tal efecto.

Una entrada de directorio es, en realidad, una pareja formada por un nombre y un número de i-nodo. En esa lista, siempre hay al menos estas dos entradas: "." (para saber su propio i-nodo) y ".." (para saber el i-nodo del padre). De esta forma se puede retroceder en el árbol de ficheros y acceder a los metadatos del directorio actual. El núcleo del sistema mantiene una caché de entradas de directorio (estructura `dentry`) en memoria para evitar estar leyendo constantemente los datos de los directorios.

Veamos un ejemplo para resolver la ruta de un fichero para poder acceder a sus datos. El fichero que se quiere leer es `/home/joe/test.txt`, se muestra el esquema en la Figura 4.5:

1. El i-nodo del raíz siempre es el 2. Para encontrar el i-nodo del directorio "home" tendremos que leer los datos del raíz y buscar la entrada, que dice que su i-nodo es el 5.
2. Se lee el i-nodo 5 del vector de i-nodos. Si no está en la caché de i-nodos, habrá que leer el bloque que contiene ese i-nodo.
3. Ahora se leen los bloques de datos de ese directorio, referenciados por su i-nodo, para buscar la entrada correspondiente a "joe". Supongamos que esa entrada se encuentra en su primer bloque de datos. Se lee el bloque 321, como indica el i-nodo.

[11] En este caso, no se hace referencia al tipo de datos que contiene el fichero; ya sabemos que en Unix los ficheros no tienen tipo de datos, el sistema es agnóstico. Se refiere a si se trata de un fichero convencional, un directorio u otro tipo de fichero no convencional.

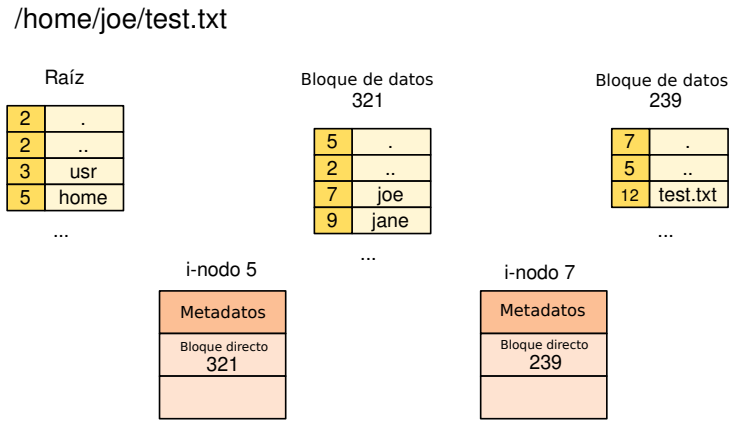

Figura 4.5: Ejemplo para resolver una ruta en Unix.

4. La entrada para "joe" dice que su i-nodo es el 7. Ahora se leerá el i-nodo 7, si no está en caché tendremos que leer el bloque correspondiente del vector de i-nodos.

5. Ahora se leen los bloques del directorio para encontrar la entrada para "test.txt". Supongamos, otra vez, que la entrada está en el primer bloque de datos. Tendríamos que leer el bloque 239.

6. La entrada para "test.txt" dice que su i-nodo es el 12. Ahora se leerá el i-nodo 12. Si no está en caché, tendremos que leer el bloque correspondiente del vector de i-nodos.

7. En el i-nodo 12 tenemos las referencias (bloques directos e indirectos de distinos niveles) para acceder a los bloques que tienen los datos de dicho fichero.

En Unix, se llama *enlace duro* cuando se da un nombre adicional a un fichero metiendo una nueva entrada en un directorio que hace referencia al mismo i-nodo. En realidad, no es diferente de una entrada de directorio normal. Un i-nodo puede tener muchas entradas apuntándole en distintos directorios. Cada vez que se crea una nueva entrada en un directorio, el campo *cuenta de referencias* del i-nodo se incrementa. Cuando se borra una entrada de directorio, se decrementa la cuenta de referencia del i-nodo correspondiente. De hecho, la llamada al sistema que sirve para eliminar un fichero se llama *unlink(2)*. Cuando la cuenta de referencias llega a cero, entonces el sistema de ficheros ya puede liberar todos sus recursos (los bloques de datos de dicho i-nodo) y marcar ese i-nodo como libre para ser reutilizado.

No tiene sentido crear un enlace duro de un fichero de un sistema de ficheros en un directorio de otro sistema de ficheros distinto, ya que los números de i-nodo sólo tienen sentido dentro de un mismo sistema de ficheros.

Tampoco se permite crear enlaces duros para directorios por dos razones fundamentales: crean bucles en el árbol de ficheros y generan ambigüedad a la hora de interpretar quién es el padre de un directorio.

Se llama *enlace simbólico*[12] (o *enlace blando*) a otra cosa distinta. Es un fichero *especial* que sirve como un puntero a otra ruta (el nombre de un fichero o directorio). En realidad es un fichero distinto cuyos datos son la ruta hacia otro fichero o directorio. Lo interpreta el sistema operativo. Al abrir el fichero, el sistema operativo se da cuenta de que es un enlace simbólico y abre de nuevo el nombre contenido en sus datos. Este tipo de ficheros se tratan de forma especial y tenemos que tener cuidado cuando operamos con ellos. La página de manual de las operaciones especificarán cómo se comportan con un enlace simbólico. Se dice que un enlace simbólico está *roto* si apunta a un fichero que no existe.

Se pueden crear enlaces simbólicos a otros sistemas de ficheros, porque en realidad lo que hacen es apuntar a una ruta. También se pueden crear enlaces simbólicos que apunten a directorios.

Veamos un ejemplo. Creamos un fichero `f1` y después creamos tres enlaces duros. Esto se hace con el comando *ln(1)*. Podemos ver con `ls -i`, que muestra el nombre y el i-nodo de cada entrada de directorio, que los cuatro nombres hacen referencia al mismo i-nodo (2368346). Con el comando `stat` podemos ver que el i-nodo tiene el contador de referencias (*links*) a cuatro:

```
 $ echo hey > f1
 $ ln f1 f2
 $ ln f2 f3
 $ ln f1 f4
 $ ls -i
2368346 f1  2368346 f2  2368346 f3  2368346 f4
 $ cat f1
hey
 $ cat f2
hey
 $ stat f1
  File: f1
  Size: 4               Blocks: 8      IO Block: 4096  regular file
Device: 805h/2053d      Inode: 2368346     Links: 4
Access: (0664/-rw-rw-r--)  Uid: ( 1000/esoriano)   Gid: ( 1000/esoriano)
Access: 2020-05-04 14:42:05.084025029 +0200
Modify: 2020-05-04 14:41:41.108620280 +0200
Change: 2020-05-04 14:41:56.028249650 +0200
 Birth: -
 $
```

En el siguiente ejemplo (continúa del anterior), se crea un enlace simbólico para `f1` llamado `e1`. Para ello, se usa el modificador `-s` del comando `ln`. Podemos ver que el i-nodo es distinto al del fichero enlazado. Ejecutamos `cat` para ver en el terminal el

[12] En Windows existe una cosa similar que recibe el nombre de *acceso directo*.

fichero enlazado. Después borramos el fichero `f1` y vemos que el enlace simbólico se ha roto:

```
 $ ln -s f1 e1
 $ ls -i e1
2368347 e1
 $ ls -l e1
lrwxrwxrwx 1 esoriano esoriano 2 may  4 14:44 e1 -> f1
 $ cat e1
hey
 $ rm f1
 $ cat e1
cat: e1: No such file or directory
 $ ls -l e1
lrwxrwxrwx 1 esoriano esoriano 2 may  4 14:44 e1 -> f1
 $
```

Parte de los metadatos de un fichero son sus permisos. Los permisos Unix son muy sencillos. Antes de verlos, tenemos que hablar un poco sobre las cuentas de usuario.

En Unix, los usuarios del sistema se describen en el fichero `/etc/passwd`. El fichero `/etc/shadow` contiene las contraseñas de los usuarios (cifradas o resumidas de alguna forma, no se mantienen en claro) y otra información sobre las credenciales. Los grupos de usuarios se describen en el fichero `/etc/group`.

Internamente, el sistema maneja dos valores enteros, el UID (***User ID***) y el GID (***Group ID***), para gestionar usuarios y grupos. Los humanos nos apañamos mejor manejando nombres. Por ejemplo, identificamos mejor a dos usuarios si hablamos "pepe" y "juan" que si hablamos de 1003 y 1043. El UID de root siempre es 0. Por eso el sistema mantiene los nombres de usuario y grupos aunque internamente sólo use los números (UID y GID). La correspondencia entre nombre de usuario y UID se mantiene en el fichero `/etc/passwd`. La de nombre de grupo y GID se mantiene en el fichero `/etc/group`.

El comando `id` nos dice nuestro UID, GID y grupos a los pertenecemos actualmente:

```
 $ id
uid=1000(esoriano) gid=1000(esoriano) groups=1000(esoriano),
4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),
128(sambashare),130(vboxusers)
 $
```

El i-nodo contiene permisos para tres entidades:

- Dueño del fichero.
- Grupo.

- El resto de usuarios del sistema.

Para cada uno de ellos se pueden asignar estos permisos:

- **r**: permiso de lectura de los datos del fichero.

  En los directorios significa que se pueden leer las entradas de directorio.

- **w**: permiso de escritura de los datos del fichero.

  En directorios significa que se pueden escribir las entradas del directorio (borrar, renombrar, añadir ficheros).

- **x**: permiso de ejecución del fichero.

  En directorios significa si se puede puede entrar o atravesar el directorio cuando se evalúa una ruta. Es necesario para acceder a un fichero del directorio (datos y metadatos).

Cuando ejecutamos `ls -l` podemos ver los permisos. El primer carácter es el tipo de fichero ("-" es un fichero convencional, "d" es un directorio, "l" es un enlace simbólico, etc.). Los siguientes 9 caracteres son los permisos (tres para dueño, tres para grupo y tres para el resto). Siempre siguen este orden: `rwx`. Si un permiso aparece como "-", entonces no está otorgado. Si aparece su letra, sí. Por ejemplo:

```
 $ ls -l /bin/cat
-rwxr-xr-x 1 root admins 35064 ene 18  2018 /bin/cat
 $
```

Ese fichero (el binario del programa `cat`) es un fichero convencional. El dueño es el usuario `root`, que puede leerlo, escribirlo y ejecutarlo. Su grupo es `admins`, y cualquier miembro de ese grupo de usuarios puede leerlo y ejecutarlo. Lo mismo para el resto de los usuarios, pueden leerlo y ejecutarlo.

Los permisos se representan normalmente en octal. Por ejemplo, 0664: 6 en binario es es 110 (por tanto `rw-`) para el dueño, lo mismo para el grupo y 4 en binario es 100 (por tanto `r--`) para el resto de los usuarios.

Podemos manipular los permisos de un fichero con el comando `chmod`. En general, eso sólo lo puede hacer el dueño del fichero y root (pero esto depende del sistema concreto). A `chmod` se le pueden dar los permisos en octal o usar modificadores como `+r` para otorgar permisos de lectura o `-x` para quitar los permisos de ejecución ("+" es otorgar, "-" es denegar). Se puede especificar también a quién se le otorga o deniega ("u" para el dueño, "g" para el grupo y "o" para el resto), por ejemplo `g+w` otorga el permiso de escritura al grupo. Por ejemplo:

```
 $ echo hey > x
 $ ls -l x
-rw-rw-r-- 1 esoriano esoriano 4 may  4 16:13 x
 $ chmod g+w x
 $ ls -l x
-rw-rw-r-- 1 esoriano esoriano 4 may  4 16:13 x
 $ chmod o-r x
 $ ls -l x
-rw-rw---- 1 esoriano esoriano 4 may  4 16:13 x
 $ chmod 700 x
 $ ls -l x
-rwx------ 1 esoriano esoriano 4 may  4 16:13 x
 $
```

Inicialmente, el creador de un fichero es su dueño y grupo. El comando `chown` cambia el dueño de un fichero. Hay que tener privilegios especiales para hacer esto[13]. El comando `chgrp` cambia el grupo de un fichero. El dueño puede cambiarlo a un grupo al que él pertenezca.

Hay otros permisos especiales:

- **sticky bit** (+t): es un permiso para directorios. Significa que no se puede borrar una entrada del directorio si no eres el dueño del directorio, el dueño del fichero o directorio que representa la entrada, o root.

  Es útil para directorios compartidos entre varios usuarios. Por ejemplo, `/tmp` tiene ese permiso porque muchos usuarios lo pueden usar para crear ficheros y directorios temporales:

  ```
   $ ls -ld /tmp
  drwxrwxrwt 59 root root 40960 may  4 16:28 /tmp
   $
  ```

- **setuid/setgid bit** (+s): el proceso que ejecute el fichero adoptará el UID/GID del dueño/grupo del fichero. Esto significa que ese proceso ejecutará a nombre del usuario dueño del fichero, no a nombre del usuario que lo ha arrancado. Esto sirve para que ciertos programas puedan elevar privilegios para realizar alguna tarea. No se suele permitir para ficheros interpretados (scripts)[14].

  Por ejemplo, el programa que cambia tu contraseña, `passwd`, necesita modificar el fichero de contraseñas, pero sólo root tiene permiso para hacer eso. Cuando un usuario normal ejecuta ese programa para cambiar su contraseña, el proceso puede

[13] En general, hay que ser root o tener alguna credencial especial.
[14] En ese caso, se suele permitir poner el bit, pero se ignora a la hora de ejecutar.

ejecutar a nombre de root para modificar la entrada del usuario en el fichero de contraseñas (`/etc/shadow`):

```
 $ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 59640 mar 22  2019 /usr/bin/passwd
 $
```

Ojo, en el caso particular de que el usuario intentando acceder al fichero sea el dueño del fichero, sólo se comprobarán los bits asociados al dueño. Al ser el dueño, puede cambiar los permisos. Puede ser sorprendente, aunque el grupo y todos tienen permiso, no puedo acceder al fichero:

```
 $ id
uid=1000(paurea) gid=1001(paurea) groups=1001(paurea),24(cdrom)
 $ ls -l uno
--w-r--r-- 1 paurea paurea 0 mar  4 10:51 uno
 $ cat uno
cat: uno: Permission denied
 $
```

Lo mismo sucede con los bits de permiso del grupo.

4.3.4. FAT vs. i-nodos

Es importante observar que las entradas de directorio de los sistemas tipo FAT contienen los metadatos de los ficheros y directorios **y el nombre**. Sin embargo. los i-nodos contienen los metadatos, pero no el nombre.

Esto tiene como principal consecuencia práctica la posibilidad de hacer enlaces duros en sistemas basados en i-nodos pero no en FAT. Tendríamos que sincronizar los tamaños y demás metadatos de todas las entradas de directorio que apuntasen a un cluster.

Otras consecuencias prácticas son las diferencias a la hora de borrar un archivo, crearlo, etc. Es importante fijarse en esas diferencias, porque afectarán al rendimiento (localidad, número de escrituras, etc.), a lo que sucederá si se estropea un bloque de disco o se va la luz, etc.

4.3.5. Gestión del espacio libre

El sistema de ficheros tiene que mantener la lista de bloques que no están siendo usados por ningún fichero. De esa lista irá cogiendo los bloques a medida que se necesitan.

Hay distintas estrategias para gestionar esto [2, 1]. Se puede usar un *bitmap* (mapa de bits). Esto es rápido, pero puede ser demasiado grande como para tenerlo entero en memoria. Por ejemplo, para un disco de 1 TB, el mapa consume 256 MB de memoria (si el bloque del sistema de ficheros es de 4KB).

Otra alternativa mantener una lista enlazada de bloques libres, pero es ineficiente si buscamos varios (pero por lo general sólo se busca uno libre). Para solucionar esto se pueden tener *agrupaciones*, de tal forma que el primer bloque de la lista no está vacío, sino que tiene enlaces a $N-1$ bloques libres, y el último al siguiente nodo de la lista. Así se pueden encontrar rápidamente las direcciones de varios bloques libres.

En las listas también se pueden tener referencias a bloques con un dato adicional: la *cuenta* de bloques adyacentes que están libres detrás de un bloque.

4.3.6. Fallos

Cuando no se *desmonta* un sistema de ficheros de forma ordenada, puede quedar corrupto. Como ya hemos visto, hay datos y metadatos en las cachés que no están sincronizados con el disco. Cuando tenemos un fallo de energía, por ejemplo, podemos llegar a tener incoherencias en el sistema de ficheros. Una pérdida de coherencia en los metadatos (estructuras del sistema de ficheros) puede ser peor que perder los datos.

El comando `fsck` sirve para reparar el sistema de ficheros después de un fallo. Ese programa recorre los bloques de los ficheros para detectar errores. Por ejemplo, mira si un mismo bloque está asignado a un i-nodo y marcado como libre (eso no puede ser). Otro ejemplo es mirar si un bloque está referenciado por dos i-nodos distintos. Estas comprobaciones son lentas y cuando se detecta un error, el usuario tampoco tiene mucho margen de maniobra (es difícil saber si un bloque dado pertenecía a un i-nodo o a otro). Los bloques huérfanos que se encuentren en estas comprobaciones se dejan tradicionalmente en el directorio `lost+found` en el raíz.

Los sistemas de ficheros modernos usan técnicas para no acabar con metadatos incoherentes, como el *journaling*. Esto consiste en *apuntar* las operaciones en una parte especial de la partición que se llama *journal*. Una vez escritas en el *journal*, las operaciones están comprometidas. En algún momento las operaciones se aplicarán en las estructuras internas del sistema de ficheros. Esto requiere ciertas garantías de que estas operaciones han llegado al disco o al menos de que llegan en determinado orden.

Si el sistema falla, al rearrancar se aplican todas las operaciones pendientes que haya en el *journal* (si las hay) para recuperar la coherencia en las estructuras internas. La desventaja es que hay que escribir los datos dos veces en el disco: en el *journal* y luego en su sitio (en las estructuras que corresponda). La ventaja es que escribir en el *journal* es menos costoso porque la escritura es secuencial. Se puede hacer *journaling* sólo de los metadatos, o de los metadatos y los datos.

En general, el *journaling* y otras estrategias de este tipo no garantizan que no perdamos datos. Lo que intentan garantizar es que el sistema de ficheros no se quede incoherente.

4.3.7. Sistemas de ficheros en Linux

En Linux, una parte del núcleo que se denomina VFS (***V**irtual **F**ilesystem **S**witch*) proporciona una interfaz común para todos los sistemas de ficheros (FAT, EXT4, etc.) [24, Capítulo 12]. Ten en cuenta que tenemos sistemas de ficheros locales (en algún

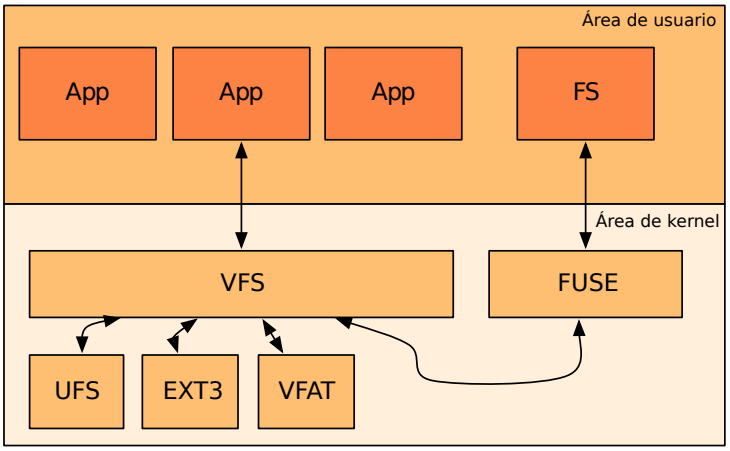

Figura 4.6: Organización de los sistemas de ficheros en Linux.

disco de nuestra máquina) y remotos. En un sistema de fichero remoto, los ficheros se encuentran en otra máquina y accedemos a ellos a través de la red.

VFS es una capa que proporciona las operaciones comunes para abrir, leer, escribir, cerrar, etc. Esta capa redirige las operaciones al sistema de ficheros concreto que la implementa.

Lo normal es que el sistema de ficheros esté implementado como un módulo del kernel, pero también puede estar implementado como un programa de espacio de usuario. En Linux podemos usar FUSE (que es un módulo del kernel) para implementar sistemas de ficheros en espacio de usuario que se puedan integrar con VFS. El esquema se muestra la Figura 4.6.

Los sistemas de ficheros estándar de Linux son los de la familia EXT (EXT2, EXT3, EXT4). Estos sistemas de ficheros implementan las ideas que hemos visto anteriormente del sistema de ficheros de Unix (i-nodos, etc.). En particular EXT4, que es el más popular ahora mismo, soporta journaling. Puedes usar el comando `dumpe2fs` para ver los detalles de una partición de tipo EXT4 (sistema de ficheros estándar de Linux).

Además, hay módulos del núcleo de Linux para usar muchos de los sistemas de ficheros de otros sistemas, como NTFS y variantes de FAT de Windows, HFS+ de Apple, etc. La página de manual *fs(5)* da información sobre los sistemas de ficheros que soporta Linux.

Nótese que, si el fichero tiene zonas grandes puestas a cero (todos los bytes en esa zona son 0x00), no hace falta reservar los bloques que cubren esa zona para almacenarlos. El sistema de ficheros se puede apuntar ese *hueco* y cuando llegue el momento de leer partes de esa zona, el sistema operativo puede *inventarse ceros* sin leerlos de ningún sitio. Es una forma de ahorrar espacio del disco. Por ejemplo, algunas herramientas de descarga crean un fichero del tamaño del fichero que se van a descargar para ir rellenándolo poco a poco por distintas partes. En este caso, el fichero recién creado sólo tiene ceros. En un sistema que soporta estos huecos, el fichero no ocupa tantos bloques como parece. A medida que se va rellenando, se van reclamando los bloques necesarios para escribir los datos. A los *ficheros con huecos* también se les llama ficheros ficheros *sparse*. No todos

los sistemas de ficheros soportan este tipo de ficheros. EXT4 sí lo hace.

4.3.8. Formateo

Cuando tenemos una partición creada, hay que *formatearla* para crear la estructura interna del sistema de ficheros que queremos usar. El formateo de una partición borrará el antiguo sistema de ficheros que tuviese la partición (si tenía alguno).

Si hemos usado alguna herramienta gráfica para crear la partición, como `gparted`, seguramente también habrá formateado la partición. Para formatear manualmente en una shell una partición con el sistema de ficheros EXT4, se usa el comando `mkfs.ext4` (ver *mkfs.ext4(8)*).

Si queremos formatear con otro sistema de ficheros, debemos usar el comando `mkfs` apropidado: `mkfs.msdos`, `mkfs.fat`, `mkfs.minix`, etc.

4.3.9. Espacio de nombres

Se denomina *espacio de nombres* a la configuración actual del árbol de ficheros. Se pueden *montar* nuevos árboles en el espacio de nombres. Montar un sistema de ficheros es como pegar un nuevo árbol en una rama del árbol actual. Por ejemplo, cuando enchufamos un *pen drive* en un sistema Linux que monte automáticamente los discos externos, vemos que podemos acceder a sus ficheros en un directorio concreto, como podría ser `/media/$USER/pendrive`. Si no se monta automáticamente, lo podemos montar manualmente. Ahora veremos cómo.

La Figura 4.7 muestra un ejemplo en el que se ha montado un sistema de ficheros que está en una unidad externa en el punto de montaje `/mnt/pen`, directorio que ya existe en el momento del montaje. A partir del montaje, ya tiene sentido una ruta como `/mnt/pen/pics`.

Las rutas tienen sentido dentro del *espacio de nombres*. Se *camina* por la ruta, resolviendo cada parte y realizando las operaciones necesarias en cada sistema de ficheros del árbol.

Tradicionalmente, en Unix hay un único espacio de nombres común para todos los procesos del sistema. Linux y otros sistemas modernos dejan tener diferentes espacios de nombres para distintos grupos de procesos[15]. Esto es, los procesos pueden tener una visión distinta del árbol de ficheros.

El comando `mount` (ver *mount(8)*) nos permite ver y modificar el espacio de nombres. Si se ejecuta sin argumentos, muestra todos los sistemas de ficheros montados. En un sistema Linux puede haber muchos sistemas de ficheros montados por omisión. Si se le pasan argumentos, sirve para montar sistemas de ficheros. El modificador `-t` sirve para indicar el tipo del sistema de ficheros que se va a montar. El modificador `-o` sirve para indicar opciones de montaje, como montar el sistema para sólo lectura, etc. Cada sistema tiene sus propias opciones, descritas en su página de manual. Por ejemplo, la página de

[15] Esta característica viene del sistema operativo Plan 9 y es en la que están basados los contenedores como Docker.

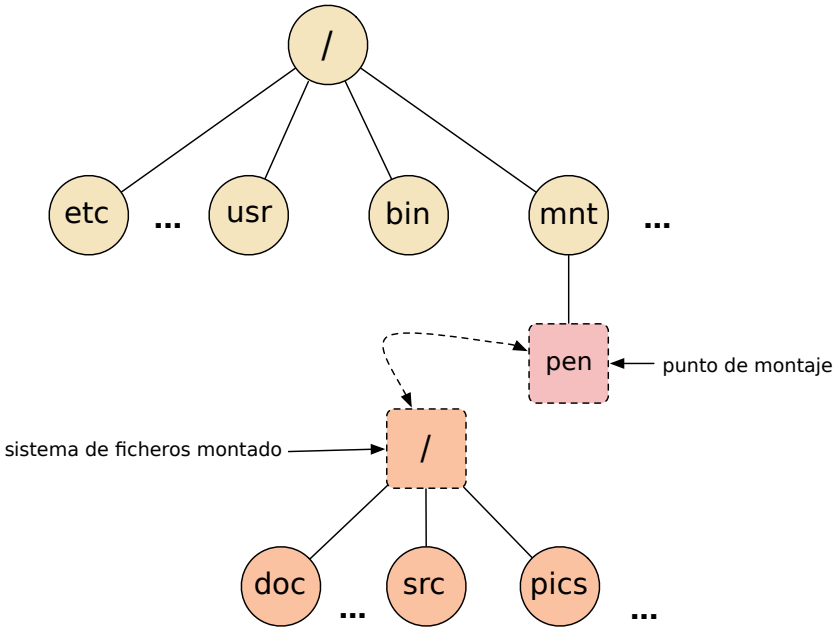

Figura 4.7: Ejemplo de espacio de nombres con dos sistemas de ficheros, el del raíz y otro montado de un pendrive.

manual para EXT4 es *ext4(5)*. Podríamos montar manualmente el sistema de ficheros del ejemplo de la Figura 4.7 así:[16]

```
$ sudo mount /dev/sdb1 -t ext4 /mnt/pen
```

Para desmontar un sistema de ficheros hay que usar el comando `umount` (ver *umount(8)*). Veamos otro ejemplo:

[16]Si no se proporciona el tipo de sistema de ficheros con `-t` se intentará detectar automáticamente.

```
 $ mount | egrep ' (ext4|vfat) ' | sed -E 's/\(.*//'
/dev/sda5 on / type ext4
/dev/sda6 on /home type ext4
/dev/sda1 on /boot/efi type vfat
 $ sudo mount /dev/sdb1 -t vfat  /mnt/externo
 $ mount | egrep ' (ext4|vfat) ' | sed -E 's/\(.*//'
/dev/sda5 on / type ext4
/dev/sda6 on /home type ext4
/dev/sda1 on /boot/efi type vfat
/dev/sdb1 on /mnt/externo type vfat
 $ umount /mnt/externo
 $ mount | egrep ' (ext4|vfat) ' | sed -E 's/\(.*//'
/dev/sda5 on / type ext4
/dev/sda6 on /home type ext4
/dev/sda1 on /boot/efi type vfat
 $
```

Listamos los sistemas de ficheros de tipo EXT4 y FAT que tenemos montados (omitiendo las opciones de montaje, que es lo que `mount` imprime entre paréntesis). Se puede ver que esta máquina tiene un sistema de ficheros EXT4 montado en el raíz y otro sistema de ficheros EXT4 montado en `/home`. También se ve que hay una partición con un sistema de ficheros FAT (es la partición de UEFI). Todos esos sistemas de ficheros están en distintas particiones del primer disco del bus SATA (`/dev/sda`).

Después, montamos un nuevo sistema de ficheros de tipo FAT de una unidad externa y volvemos a listar los montajes como antes. Se ve que el disco externo es `/dev/sdb`. Al final desmontamos el sistema de ficheros y volvemos a listar.

El fichero de configuración `/dev/fstab` contendrá una línea por cada sistema de ficheros que se monte por defecto en el sistema (ver *fstab(5)*).

4.4. Dispositivo de loopback e imágenes de disco

Vimos en la sección 4.2.2 lo que era una imágen de disco. En Linux, estas imágenes se pueden montar igual que si fuesen un disco duro, utilizar el sistema de ficheros que contienen y desmontarlos.

El dispositivo de *loop*, llamado dispositivo de *loopback*, permite crear un dispositivo como si de un disco duro se tratara, asociado a un fichero que puede contener un sistema de ficheros, particiones, etc. A continuación se puede montar. El siguiente ejemplo crea un fichero de 10 megabytes, lo formatea con `mkfs.ext4`, lo monta utilizando el dispositivo de loopback, crea un fichero dentro y lo desmonta:

```
 $ cd /tmp
 $ dd if=/dev/zero bs=10M count=1 of=disk
1+0 records in
1+0 records out
10485760 bytes (10 MB, 10 MiB) copied, 0,0142014 s, 738 MB/s
 $ mkfs.ext4 disk
mke2fs 1.45.5 (07-Jan-2020)
Discarding device blocks: done
Creating filesystem with 2560 4k blocks and 2560 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

 $  sudo mkdir prueba
 $  sudo mount -o loop disk /tmp/prueba
# Esto equivale a mount /tmp/disk /tmp/prueba -t ext4 -o loop=/dev/loop3
#         suponiendo que loop3 está libre
#        (mount busca uno libre automáticamente)
#         si se hiciese manualmente loop3 se apunta al
#         fichero disk mediante el comando losetup previamente
#                 losetup /dev/loop3 /tmp/disk
# Buscar uno automáticamente como hace mount sería:
#                 losetup -f /tmp/disk
 $  sudo touch prueba/xxx
 $  sudo ls prueba
lost+found  xxx
 $  sudo umount prueba
 $
```

4.5. Usando los ficheros

Esta sección se centra en el uso de los ficheros usando la interfaz de llamadas al sistema y funciones de biblioteca.

4.5.1. Procesos y ficheros

Por cada proceso, el kernel mantiene una tabla con los ficheros que tiene abiertos. Esto lo hace usando un atributo del proceso que se llama **tabla de descriptores de fichero**.

Como vimos en el Capítulo 2, un fichero abierto tiene un *descriptor de fichero* (*file descriptor*) que lo identifica. Esa estructura tiene la posición actual del fichero (*offset*),

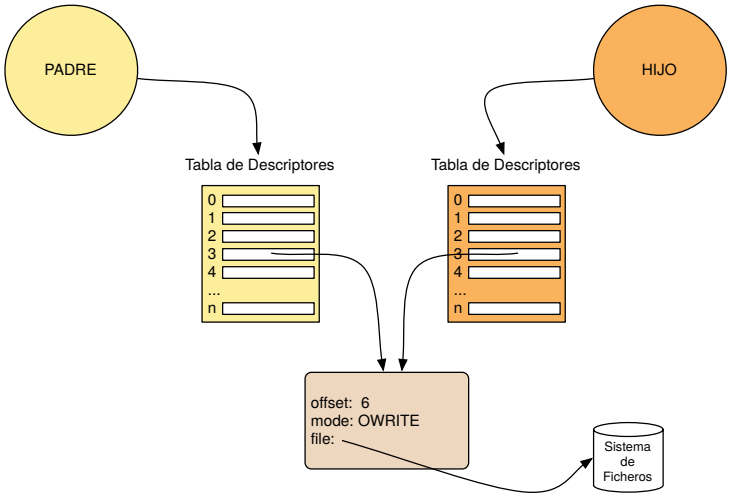

Figura 4.8: Ficheros abiertos padre e hijo: el hijo tiene su propia tabla, que apuntará a estructuras compartidas con el padre en el caso de los ficheros que ya estaban abiertos cuando se llamó a `fork`.

esto es, la posición del fichero (el byte) por la que vamos leyendo o escribiendo. También tiene el modo de apertura del fichero (lectura, escritura, etc.).

El número de descriptor de fichero es la posición en dicha tabla. Cuando se abre un fichero, se usa la primera entrada de la tabla que esté sin usar.

Cuando se crea un proceso, el hijo hereda una copia exacta de la tabla del padre, compartiendo las estructuras que representan los ficheros ya abiertos **en el momento de creación**. Esto significa que las operaciones del padre afectan el offset que ve el hijo, y viceversa. Pero atención: esto sólo pasa con los ficheros que ya están abiertos en el momento de la creación del proceso. A partir del momento de creación, cada proceso colocará en su tabla las estructuras de los ficheros que abra, sin compartir nada. La Figura 4.8 muestra un esquema.

Hay tres posiciones especiales en la tabla: (0) entrada estándar, (1) salida estándar, y (2) salida de errores. Un convenio importante en Unix es que estos tres descriptores siempre tienen que estar abiertos y listos para su uso. Cuando un programa comienza a ejecutar, siempre tiene que tener disponibles estos descriptores. El padre del proceso se tiene que encargar de dejar la tabla con estas entradas listas para usarse (y cerrar el resto de entradas para dejarla limpia).

No hay que confundir entrada estándar o salida estándar con el terminal. Los procesos que creamos en la shell normalmente tienen como entrada, salida y salida de errores apuntando al **terminal**. Esto no significa que no se pueda crear un proceso cuya entrada estándar, salida estándar y/o salida de errores sea cualquier otro fichero. En el Capítulo 2 ya hemos visto las redirecciones.

Cuando un programa crea un proceso mediante *fork(2)*, la entrada, salida y salida de errores del hijo serán las mismas que las del padre, ya que el hijo tiene una copia de la tabla del padre y esos ficheros ya estaban abiertos. Ya veremos cómo podemos

cambiarlas haciendo redirecciones desde nuestro programa.

El terminal es un fichero como cualquier otro (un fichero virtual). Cuando se lee de ese fichero, se lee del teclado. Cuando se escribe en ese fichero, se escribe en la pantalla. Un proceso siempre se puede referir a su terminal con la ruta `/dev/tty` (esa ruta se resuelve, para cada proceso, como su terminal, ver *tty(4)*). En realidad, el terminal será otro fichero (p. ej. `/dev/pts/31` o `/dev/tty3`). Para saber en qué terminal estamos trabajando, podemos ejecutar el comando `tty`:

```
 $ tty
/dev/pts/1
 $
```

En `/proc` podemos inspeccionar cuáles son los ficheros que tiene abiertos un proceso en su tabla. Por ejemplo:

```
 $ sleep 1000 < /dev/null > /tmp/a
^Z
[1]+  Stopped                 sleep 1000 < /dev/null > /tmp/a
 $ bg
[1]+ sleep 1000 < /dev/null > /tmp/a &
 $ ps
  PID TTY          TIME CMD
17154 pts/1    00:00:00 bash
27772 pts/1    00:00:00 sleep
27773 pts/1    00:00:00 ps
 $ ls -l /proc/27772/fd
total 0
lr-x------ 1 esoriano esoriano 64 may  4 18:35 0 -> /dev/null
l-wx------ 1 esoriano esoriano 64 may  4 18:35 1 -> /tmp/a
lrwx------ 1 esoriano esoriano 64 may  4 18:35 2 -> /dev/pts/1
 $
```

El ejemplo ejecuta un `sleep`[17] redirigiendo su salida al fichero `/tmp/a` y su entrada a `/dev/null`, no se toca la salida de errores (por tanto se queda con la que hereda de la shell, el terminal). Paramos el proceso con `Ctrl+z` y lo mandamos a segundo plano. Después vemos en `/proc` los ficheros que tiene abierto el proceso (cuyo PID es 27772). Vemos que coincide con las redirecciones que hemos hecho en la shell (`/dev/pts/1` es el terminal que usa ese proceso). En el directorio `fd` del directorio que representa el proceso en `/proc`, tenemos enlaces simbólicos a cada uno de los ficheros abiertos. El nombre del enlace simbólico es el índice en la tabla de procesos (0 para la entrada estándar, etc.).

[17] El comando `sleep` no lee nada de la entrada ni escribe nada en su salida, se usa en este ejemplo porque al quedarse dormido esos segundos nos pemite inspeccionar sus descriptores tranquilamente.

4.5.2. Leyendo y escribiendo

La llamada al sistema *read(2)* lee como mucho el número de bytes indicado del descriptor (que representa un fichero abierto) que se le pasa como primer parámetro. Los datos se copian en memoria a partir de la dirección especificada por el segundo parámetro. Se podrán leer hasta el número de bytes especificado en el tercer parámetro. La función retorna el número de bytes que se han leído del fichero y se han copiado a la memoria (o un error).

```
ssize_t read(int fd, void *buf, size_t count);
```

La llamada `read` puede leer menos bytes que los que se han requerido sin que esto sea un error. Esto se llama **lectura corta**. Si se desea leer más bytes, habrá que realizar otra llamada. Esto puede ocurrir por distintos motivos: no es asunto del programa que realiza la llamada.

Hay que tener claro que el contrato que se tiene con `read` es el siguiente:

- Puede retornar cualquier número de bytes desde 1 hasta los que se han solicitado. Que devuelva menos bytes que los que se han pedido **no significa que haya habido un error o que no haya más datos en el fichero**. Simplemente se trata de una *lectura corta*.
- Si devuelve 0 bytes, se ha llegado al final del fichero. No hay más datos que leer.
- Si devuelve -1, es que ha habido un error, y la variable `errno` tendrá más información. Los detalles sobre la llamada están en su página de manual (*read(2)*).

La llamada al sistema *write(2)* **es distinta**, aunque su cabecera sea similar:

```
ssize_t write(int fd, const void *buf, size_t count).;
```

Esta llamada escribe el número indicado de bytes del buffer, en el fichero abierto que indica el primer parámetro. Los datos a escribir deben estar en la dirección pasada en el segundo parámetro (el buffer). Escribirá tantos bytes como indique el tercer parámetro.

La escritura no se comporta como la lectura: si `write` devuelve un número distinto al número de bytes solicitado en el tercer parámetro, se debe considerar un error. Esto es fácil de entender con un ejemplo de la vida real. Si me dan 10 botellas de leche para llevar a casa de mi hermana (`write`), no llevarlas o llegar con sólo 3, es un error: ha pasado algo malo. Si me piden ir a por 10 botellas de leche al supermercado (`read`), puede que sólo queden 3. No es un error, habrá que volver otro día a por más. En un caso, parto con las botellas en la mano y en el otro, no sé cuántas hay. Hay una incertidumbre que hay que tratar en la interfaz.

En ambos casos, se leerá o escribirá desde la posición actual del fichero, que se llama *offset*. Cada vez que leemos o escribimos, se modifica el *offset* del descriptor de fichero, va avanzando a medida que hacemos operaciones.

Cuando una lectura llega al final del fichero, devuelve los datos que ha leído. La siguiente lectura retornará 0 ya que no hay más datos en el fichero (suponiendo que no ha crecido, claro).

En una escritura, si el número de bytes que se quieren escribir es mayor que la constante `SSIZE_MAX` (definida en la libc) el comportamiento de `write` está por definir. El standard de POSIX lo fija a un tamaño pequeño (32 KB):

```
1 #define _POSIX_SSIZE_MAX          32767
```

Sin embargo, Linux lo establece a:

```
1 #define SSIZE_MAX          LONG_MAX
```

Por tanto, $2^{31} - 1 = 2GB$.

Las escrituras sobrescriben los datos en la zona correspondiente. Si hacemos una escritura de 100 bytes desde el offset 1000, los bytes del 1000 al 1099 se han sobrescrito, y el offset actual será 1100.

El fichero crece si escribimos a partir del final. Si hubiese un hueco (porque se haya movido el *offset* más allá del final, como se verá más adelante) se rellena con ceros.

El siguiente programa lee de la entrada estándar hasta que se agota. Todo lo que lee, lo escribe por la salida estándar:

Programa 4.1: read.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <err.h>
6  #include <unistd.h>
7
8  enum {
9      Bufsize = 8 * 1024,
10 };
11
12 int
13 main(int argc, char *argv[])
14 {
15     char buf[Bufsize];
16     int nr;
17
18     while ((nr = read(0, buf, Bufsize)) != 0) {
19         if (nr < 0) {
20             err(EXIT_FAILURE, "can't read");
21         }
22         if (write(1, buf, nr) != nr) {
23             err(EXIT_FAILURE, "can't write");
24         }
25     }
26     exit(EXIT_SUCCESS);
27 }
```

El programa usa un *buffer* de 8 KB para realizar las lecturas. Por cada lectura, se hace una escritura de lo leído en la salida estándar:

```
 $ echo hey | ./read
hey
 $  ./read < quijote.txt | wc -l
74294
 $ wc -l quijote.txt
74294 quijote.txt
 $ ./read
hey
hey
ho
ho
lets go
lets go
^D
 $
```

¿Cuántas llamadas a `read` han hecho falta para cada ejecución? No lo sabemos. En el primer caso, seguramente una lectura es capaz de leer los datos y la segunda habrá devuelto cero. En el segundo, muchas porque el fichero `quijote.txt` es grande. En este

caso, seguramente todas las llamadas menos la última hayan leído 8 KB. En la última, todas las lecturas han sido cortas, porque la entrada estándar es el terminal: cada línea escrita en el terminal (representada en naranja) se ha leído en un `read`[18]. Cuando se ha pulsado `Ctrl+d`, la entrada termina y `read` retorna 0. El programa lee y escribe del terminal, por eso vemos la líneas repetidas (en naranja las que escribimos y en negro las que escribe el programa).

Hay versiones de `read` y `write` a las que se le pasa el offset en el que deben operar: *pread(2)* y *pwrite(2)*. Para estas llamadas no se usa el offset que mantiene el descriptor de fichero, se usa el que se le pasa a la función en el cuarto parámetro. Estas llamadas no se suelen usar mucho porque casi siempre será más cómodo no lidiar con el offset, pero pueden resultar útiles para algunos programas:

```
ssize_t pread(int d, void *buf, size_t nbyte, off_t offset);
```

```
ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t offset);
```

Para manipular el offset del descriptor tenemos la llamada al sistema *lseek(2)*:

```
off_t lseek(int fd, off_t offset, int whence);
```

Esta llamada cambia el offset del descriptor de fichero. Retorna el offset resultante después de la modificación, -1 si hay algún error. Tiene tres modos de operación (que se especifica con estas constantes en el tercer parámetro):

- `SEEK_SET`: el offset es absoluto.
- `SEEK_CUR`: se suma al offset actual.
- `SEEK_END`: se suma al offset del final del fichero.

Para crear un fichero de un tamaño dado puesto a ceros, no es necesario escribir ceros hasta llegar al tamaño. Es suficiente con crear el fichero, posicionar el offset con `lseek` en el tamaño deseado menos un byte y escribir un único byte (0x00). Esto además permite crearlo *sparse* (fichero con huecos). El sistema de ficheros no necesita rellenar realmente los huecos con ceros, sino que se los "inventa" cuando se lee. Así el fichero ocupa menos espacio real de disco.

[18] Es como se comporta el terminal: cada línea provoca una lectura (también un Ctrl+d si quedan datos por entregar).

4.5.3. Crear, abrir y cerrar ficheros

Para trabajar con ficheros que no son heredados (como sí lo son la entrada estándar, salida estándar y salida estándar de errores), debemos abrirlos antes. Para ello se usa la llamada al sistema *open(2)*.

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

La llamada `open` abre un fichero en el modo indicado por el segundo parámetro. El descriptor ocupará la primera posición en la tabla que esté libre. Retorna el número del descriptor, -1 en error.

El segundo parámetro es una combinación de *flags* que indican el modo en el que queremos operar con el fichero. Las flags más importantes son (todas están descritos en *open(2)*):

- `O_CREAT`: si el fichero no existe, se debe crear. En este caso, hay que proporcionar el tercer parámetro. Ese parámetro son los permisos del fichero. Como se puede observar, es un entero. Lo más cómodo es pasar los permisos en octal como hemos visto anteriormente. Recuerda que en C un literal octal empieza por cero (p. ej. 0664).
- `O_RDONLY`: el fichero sólo se va a leer.
- `O_WRONLY`: el fichero sólo se va a escribir.
- `O_RDWR`: el fichero se va a leer y escribir.
- `O_TRUNC`: si el fichero existe y se abre para escribir (`O_RDWR` o `O_WRONLY`), entonces se trunca a longitud cero. Se descartan los datos que tuviese el fichero antes de la apertura. Se suele usar junto con `O_CREAT` para actuar sobre un fichero limpio (si existe se trunca y si no existe se crea).
- `O_APPEND`: se va a escribir siempre al final del fichero. Cualquier escritura que se haga en el fichero es como si se hiciera un `lseek` al final del fichero seguido de un `write`[19].
- `O_CLOEXEC`: el fichero se cierra automáticamente si se llama a `exec`. Esto es una medida de seguridad para que otro programa no acceda a los ficheros que usa el programa que abre el fichero.

Estas flags son bits en un entero y se conjugan usando la operación *or* bit a bit. Por ejemplo, para crear o truncar el fichero si existe y leer de él usaríamos la combinación:

[19] Y esto se hace de forma *atómica*, ya hablaremos de eso en el Capítulo 7.

`O_WRONLY|O_CREAT|O_TRUNC`

Ojo, los modos de apertura no se pueden combinar. Esta combinación es incorrecta: `O_WRONLY|O_RDONLY|O_CREAT|O_TRUNC` y se escribiría: `O_RDWR|O_CREAT|O_TRUNC`.

El control de acceso (esto es, la comprobación de permisos) se realiza en la apertura. La comprobación y la reserva y liberación de recursos son las razones para la existencia de las operaciones `open` y `close`). Una vez abierto, ya podremos trabajar con el fichero. El offset del descriptor de fichero será 0 (estamos posicionados en el byte 0 del fichero). Es importante tener en cuenta que si el proceso abre dos veces el mismo fichero, son dos descriptores de fichero diferentes (cada uno con su offset, modo de apertura, etc.).

Si se crea el fichero con `O_CREAT`, el dueño del fichero será el usuario a nombre del que ejecuta el proceso. Dependiendo del sistema, el grupo puede ser el GID del proceso (en sistemas Unix derivados de System V) o el grupo del directorio en el que se crea (en sistemas BSD). En Linux es una mezcla: hace una u otra cosa dependiendo de si el directorio en el que se crea no tenga o sí tenga el bit `setgid`.

Cuando se crea el fichero, se queda abierto en el modo indicado, independientemente de los permisos que se le asignen en su tercer parámetro.

Programa 4.2: forkwrite.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <unistd.h>
9  #include <err.h>
10
11 int
12 main(int argc, char *argv[])
13 {
14     int fd;
15
16     switch (fork()) {
17     case -1:
18         err(EXIT_FAILURE, "can't fork");
19     case 0:
20         fd = open("/tmp/a", O_WRONLY | O_CREAT | O_TRUNC, 0666);
21         if (write(fd, "son\n", 4) != 4) {
22             err(EXIT_FAILURE, "can't write");
23         }
24         break;
25     default:
26         fd = open("/tmp/a", O_WRONLY | O_CREAT | O_TRUNC, 0666);
27         if (write(fd, "father\n", 7) != 7) {
28             err(EXIT_FAILURE, "can't write");
29         }
30     }
31     close(fd);
32     exit(EXIT_SUCCESS);
33 }
```

Si ejecutamos ese programa:

```
 $ ./forkwrite
 $ cat /tmp/a
son
 $
```

¿Qué ha pasado? Padre e hijo crean/truncan el fichero. Como lo abren por separado, cada uno tiene un descriptor distinto. Los dos comienzan con el offset 0 y escriben desde el principio del fichero (no comparten offset). Como no sabemos cómo se ordenarán las instrucciones del padre y del hijo, ya podemos imaginar que este programa puede acabar dejando el fichero con la línea "son\n" o con la línea "father\n". Si acaba el padre antes de empezar el hijo, el resultado sería:

```
 $ ./forkwrite
 $ cat /tmp/a
father
 $
```

Pero el asunto es más complicado todavía. Imagina que se ordenan así:

1. El padre crea el fichero.
2. El hijo trunca el fichero (ya estaba vacío).
3. El padre escribe "father\n" desde el offset 0.
4. El hijo escribe "son\n" desde el offset 0.
5. El fichero quedaría con dos líneas: "son\ner\n" resultado del solapamiento de los dos writes.

El resultado sería:

```
 $ ./forkwrite
 $ cat /tmp/a
son
er
 $
```

Cualquiera de los resultados de las posibles ordenaciones (como las tres que hemos visto, "son\n", "son\ner\n" o "father\n") puede suceder en una ejecución concreta. Puedes experimentar con el programa, poniendo llamadas a *sleep(3)* después de las llamadas a `open` para forzar esta secuencia de operaciones, y comprobar el resultado.

La concurrencia nos sorprende muchas veces. Es difícil de entender sus efectos. Las escrituras en un mismo fichero convencional desde distintos procesos concurrentes con

distintos descriptores de fichero son imprevisibles: no hay ninguna garantía. El estándar POSIX dice que el comportamiento es indefinido y que se debe evitar. Si es necesario que varios procesos trabajen a la vez con un fichero, debemos sincronizarlos para evitar errores. En el Capítulo 7 trataremos este asunto.

Supongamos que cambiamos el programa de esta forma:

Programa 4.3: forkwrite2.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <unistd.h>
9  #include <err.h>
10
11 int
12 main(int argc, char *argv[])
13 {
14     int fd;
15
16     fd = open("/tmp/a", O_WRONLY | O_CREAT | O_TRUNC, 0666);
17     if (fd < 0) {
18         err(EXIT_FAILURE, "can't open");
19     }
20     switch (fork()) {
21     case -1:
22         err(EXIT_FAILURE, "can't fork");
23     case 0:
24         if (write(fd, "son\n", 4) != 4) {
25             err(EXIT_FAILURE, "can't write");
26         }
27         break;
28     default:
29         if (write(fd, "father\n", 7) != 7) {
30             err(EXIT_FAILURE, "can't write");
31         }
32     }
33     close(fd);
34     exit(EXIT_SUCCESS);
35 }
```

Si lo ejecutamos:

```
 $ ./forkwrite
 $ cat /tmp/a
son
father
 $
```

Ahora comparten el descriptor de fichero, porque el fichero se abre antes de crear el hijo. Por tanto, ahora comparten offset y no se pisarán. Linux, desde su versión 3.14, garantiza que dos escrituras que comparten descriptor de fichero no se solaparán, como

dicta el estándar POSIX.1-2008[20]. Veremos un mensaje y después el otro. No sabemos en qué orden, podríamos ver antes la escritura del padre y después la del hijo (o al revés).

Cuando se crea un fichero, sus permisos finales dependen de la **máscara de creación (`umask`)**. Esta máscara evita que se creen ficheros y directorios con permisos inadecuados. Es una propiedad del proceso (como su PID o el directorio de trabajo actual) y se hereda del padre. Es un valor entero: los bits puestos a 1 (i.e. los permisos activados) en esa máscara no se pondrán en los permisos del fichero cuando se cree, independientemente de qué diga el tercer parámetro de `open`. Los permisos finales del fichero serán puestos al valor indicado por la siguiente expresión (siendo `mode` el modo que hemos pasado como tercer parámetro a `open`):

```
(mode & ~umask)
```

Por ejemplo, si `umask` es 0077, aunque el tercer parámetro de `open` sea 0755, los permisos que se le ponen al fichero son 0700. Si queremos saber o cambiar el valor de la máscara en la shell, podemos usar el comando `umask`:

```
$ umask
0002
$
```

Por tanto, si se crea un fichero con `open`, nunca tendrá el permiso 0002 puesto porque lo impide la máscara. El valor 0002 es 10 en binario (esto es, `--- --- -w-`). Por tanto se está denegando el permiso "w" para los usuarios que no son el dueño o del grupo del fichero.

Desde un programa podemos controlar la máscara con la llamada al sistema *umask(2)*, que pone la máscara al valor pasado en su parámetro y devuelve el valor que tenía anteriormente:

```
mode_t umask(mode_t mask);
```

4.5.4. Redirecciones

Como ya sabemos después de usar la shell, las redirecciones son muy útiles. La llamadas al sistema `dup` y `dup2` permiten realizar redirecciones en un programa[21].

[20] Ver *write(2)*, sección *BUGS*.

[21] En realidad se puede hacer con `close` y `open` (`open` asigna el menor número descriptor libre). Esto tiene el problema de que depende de que no hayan cerrado un descriptor más pequeño y además no

La llamada al sistema *dup(2)* duplica un descriptor de fichero en la primera posición disponible en la tabla. Retorna el número del descriptor en el que se ha duplicado y -1 en caso de error. La llamada *dup2(2)* duplica el descriptor en la posición indicada en el segundo parámetro, cerrándolo antes si estaba abierto.

```
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

Este programa redirige su salida estándar al fichero que se le pasa como argumento (lo crea si no existe y lo trunca si existe) y después escribe en su salida:

Programa 4.4: dup.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <errno.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <err.h>
9
10 int
11 main(int argc, char *argv[])
12 {
13     int fd;
14
15     if (argc != 2) {
16         errx(1, "usage: %s file", argv[0]);
17     }
18     fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0640);
19     if (fd < 0) {
20         err(1, "can't open file");
21     }
22     dup2(fd, 1);
23     close(fd);
24     printf("hey hey hey!!!\n");
25     exit(EXIT_SUCCESS);
26 }
```

Si lo ejecutamos:

```
$ ./dup /tmp/x
$ cat /tmp/x
hey hey hey!!!
$
```

es atómico. En general es mejor utilizar `dup2` que `open` o `dup`, porque `dup2` expresa claramente lo que se quiere hacer y confina los efectos laterales de la llamada (otras no pueden interferir con ella ejecutando antes o en otros threads).

Cuando hemos abierto un fichero para trabajar con él y ya no lo necesitamos más, lo debemos cerrar. La llamada *close(2)* cierra un fichero abierto y libera los recursos asociados.

```
int close(int fd);
```

Recuerda que las tres primeras posiciones en la tabla de descriptores no se deben quedar cerradas: el proceso siempre debe tener entrada, salida y salida de errores.

4.5.5. Consultar metadatos

La llamada al sistema *access(2)* permite comprobar si tenemos permisos para ejecutar, leer o escribir un fichero. Devuelve 0 si se puede acceder al fichero en el modo indicado por su único parámetro, -1 en otro caso. Cuidado, no es una función booleana (no interpretes el valor devuelto como un booleano).

```
int access(const char *pathname, int mode);
```

Tenemos las siguientes constantes para el parámetro. Las tres últimas se puede conjugar con un *or* bit a bit:

- `F_OK`: el fichero/directorio existe.
- `R_OK`: se puede leer.
- `W_OK`: se puede escribir.
- `X_OK`: se puede ejecutar.

Se puede elegir entre `F_OK` o una combinación de las otras tres (por ejemplo `W_OK|X_OK`) pero no se pueden mezclar. Esta combinación: `F_OK|X_OK`, es un error.

Para consultar todos los metadatos de un fichero tenemos la llamada al sistema *stat(2)*:

```
int stat(const char *pathname, struct stat *statbuf);
```

Esta llamada lee los metadatos de un fichero (que están en su estructura i-nodo) y los escribe en la estructura de tipo `struct stat` que se pasa por referencia. Si falla, retorna -1. Cuidado, tanto `access` como `stat` atraviesan los enlaces simbólicos. Si queremos los metadatos del enlace y no del fichero apuntado, hay que usar *lstat(2)*.

La estructura de tipo `struct stat` tiene bastantes campos. Se puede consultar toda la información en las páginas de manual *stat(2)* e *inode(7)*.

Los más importantes son:

- `st_ino`: i-nodo del fichero.
- `st_mode`: entero con el *modo*, que son permisos, etc.
- `st_nlinks`: número de nombres (enlaces duros).
- `st_uid`: UID del dueño.
- `st_gid`: GID del grupo.
- `st_size`: tamaño del fichero en bytes.
- `st_atime`: fecha del último acceso al fichero.
- `st_mtime`: fecha de la última modificacón de sus datos.
- `st_ctime`: fecha del último cambio en sus datos o metadatos (excepto en el tiempo de acceso).

Tradicionalmente, las fechas en Unix se representan como número de segundos transcurridos desde el 1 de enero de 1970. Esta representación de tiempo se conoce como *Unix epoch*, *Unix time* o *POSIX time*. Los sistemas modernos siguen representando el tiempo como unidades desde dicha fecha, pero ya no lo cuentan en segundos porque es un grano demasiado grueso. Desde hace tiempo, Linux usa nanosegundos para las fechas.

El campo `st_mode` es un valor entero que codifica en sus bits distintos metadatos del fichero. Por ejemplo, los 9 bits de sus permisos son parte de este entero. Para consultar los metadatos embebidos en ese valor, tenemos definidas distintas constantes.

Por ejemplo, para saber si se trata de un fichero, directorio, enlace simbólico, etc. podemos aplicar la máscara `S_IFMT` para quedarnos sólo con los bits de interés (los que indican el tipo de fichero) haciendo un *and* bit a bit y después comparar con las constantes:

- `S_IFDIR`: es un directorio.
- `S_IFREG`: es un fichero normal.
- `S_IFLNK`: es un enlace simbólico.

Las máscaras están definidas en la página de manual *inode(7)*. Por ejemplo, para comprobar si se trata de un fichero convencional, podemos hacerlo así:

```
1 if(stat(path, &st) < 0){
2       ... // handle error
3 }
4 if((st.st_mode & S_IFMT) == S_IFREG){
5       printf("it's a regular file!\n");
6 }
```

También se pueden usar macros para consultar el tipo (descritas en *inode(7)*). Por ejemplo, esto hace lo mismo que lo anterior:

```
1 if(stat(path, &st) < 0){
2       ... // handle error
3 }
4 if(S_ISREG(st.st_mode)){
5       printf("it's a regular file!\n");
6 }
```

Para consultar los permisos tenemos otras constantes definidas `S_IRUSR` es el bit "r" para el dueño, `S_IRUSW` es el bit "w" para el dueño, etc. Los 9 bits de los permisos son los bits de menos peso del campo. Aplicando un *and* bit a bit podemos saber si el fichero tiene el permiso que queremos consultar.

Veamos un ejemplo. Este programa mira si el argumento que se le pasa es un fichero convencional o un directorio. Si lo es, imprime su dueño y sus permisos:

Programa 4.5: stat.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int
isfileordir(struct stat *st)
{
    int isf;

    isf = (st->st_mode & S_IFMT) == S_IFREG;
    return isf || (st->st_mode & S_IFMT) == S_IFDIR;
}

int
main(int argc, char *argv[])
{
    struct stat st;
    unsigned int perms;

    if (argc != 2) {
        errx(EXIT_FAILURE, "usage: %s file", argv[0]);
    }
    if (stat(argv[1], &st) < 0) {
        err(EXIT_FAILURE, "stat failed");
    }
    perms = ((unsigned long)st.st_mode) & 0777;
    if (isfileordir(&st)) {
        printf("size: %ld\nowner: %d\nperms: 0%o\n",
               st.st_size, st.st_uid, perms);
    } else {
        errx(EXIT_FAILURE,
             "%s is not a regular" "file or a directory", argv[1]);
    }
    exit(EXIT_SUCCESS);
}
```

Si lo probamos:

```
 $ ls -l > /tmp/afile
 $ ./stat  /tmp/afile
size: 1175
owner: 1000
perms: 0664
 $ mkdir /tmp/adir
 $ ./stat  /tmp/adir
size: 4096
owner: 1000
perms: 0775
 $ ./stat  /dev/tty1
stat: /dev/tty1 is not a regular file or a directory
 $
```

Para modificar los metadatos de un fichero tenemos distintas llamadas al sistema, por ejemplo:

- *chmod(2)*: cambia los permisos.
- *chown(2)*: cambia el dueño y grupo.
- *utime(2)*: cambia la fecha de acceso y modificación.

4.5.6. Directorios

Los directorios tienen sus propias operaciones de lectura y escritura. Ya sabemos que un directorio es una lista de entradas de directorio, y una entrada de directorio es una pareja formada por el nombre y el i-nodo correspondiente. Bueno, en realidad las entradas de directorio pueden tener más cosas, dependiendo del sistema (es algo dependiente de la implementación).

En Linux una entrada de directorio está definida por esta estructura:

```
1  struct dirent {
2        ino_t          d_ino;       /* Inode number */
3        off_t          d_off;       /* Not an offset; see below */
4        unsigned short d_reclen;    /* Length of this record */
5        unsigned char  d_type;      /* Type of file; not supported
6                                       by all filesystem types */
7        char           d_name[256]; /* Null-terminated filename */
8    };
```

Además del nombre y el i-nodo, se almacena el tipo de fichero (convencional, directorio, enlace, etc.) y otras cosas. Si queremos hacer programas portables entre distintos sistemas de tipo Unix, no debemos usar esos campos extra. El campo `d_type` es útil cuado estamos

recorriendo directorios (nos ahorramos una llamada a `stat` para saber de qué tipo es una entrada), pero si lo usamos estamos haciendo nuestro programa menos portable[22].

Los directorios se abren con *opendir(2)*. Esta llamada abre un directorio para lectura. El directorio abierto se representa con la estructura de tipo `DIR` que nos devuelve la función (nos devuelve un puntero a ese tipo). En caso de error devuelve NULL.

```
DIR *opendir(const char *dirname);
```

Una vez abierto el directorio, podemos ir leyendo entrada a entrada usando *readdir(2)*.

```
struct dirent *readdir(DIR *dirp);
```

La función retorna un puntero a la siguiente entrada de directorio, representada por la estructura `dirent` que hemos visto anteriormente. Retorna NULL en caso de que no haya más entradas o ocurra un error.

Siempre hay que cerrar el directorio que hemos abierto para leer. Esto se hace con *closedir(2)*, pasando el puntero a DIR con el que estábamos trabajando. Si no cerramos el directorio, no se liberan los recursos que se han reservado para leer el directorio entrada a entrada (tenemos un *leak* de memoria, además de dejar cosas abiertas).

```
int closedir(DIR *dirp);
```

Por ejemplo, este es un `ls` simplificado:

[22] Otros sistemas de tipo Unix pueden no tener ese campo, o tenerlo y que se llame de forma diferente, o que contenga otra información, etc.

Programa 4.6: minils.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <err.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10 #include <dirent.h>
11
12 int
13 main(int argc, char *argv[])
14 {
15     DIR *d;
16     struct dirent *ent;
17
18     if (argc != 2) {
19         errx(EXIT_FAILURE, " usage: %s dir", argv[0]);
20     }
21     d = opendir(argv[1]);
22     if (d == NULL) {
23         err(EXIT_FAILURE, "opendir failed: %s", argv[1]);
24     }
25     while ((ent = readdir(d)) != NULL) {
26         if (ent->d_name[0] != '.') {
27             printf("%s\n", ent->d_name);
28         }
29     }
30     closedir(d);
31     exit(EXIT_SUCCESS);
32 }
```

Si lo probamos:

```
 $ mkdir /tmp/dir
 $ echo hi > /tmp/dir/afile
 $ echo ho > /tmp/dir/anotherfile
 $ ls /tmp/dir
afile  anotherfile
 $ ./myls /tmp/dir/
anotherfile
afile
 $
```

No se permite escribir los directorios directamente. Tenemos una operación para crear directorios llamada *mkdir(2)*:

```
int mkdir(const char *path, mode_t mode);
```

Esta llamada crea un directorio con la ruta especificada. El segundo parámetro son sus permisos. Igual que en el caso de `open`, los permisos están sujetos al valor de la máscara de creación `umask`.

Hay distintas llamadas al sistema para manipular las entradas de directorio. Por ejemplo, la llamada *rename(2)* renombra o mueve un fichero (dependiendo si el directorio origen y el directorio destino son el mismo o no). Para eliminar una entrada de directorio debemos usar *unlink(2)*, excepto si queremos borrar un directorio, en ese caso debemos usar *rmdir(2)*.

La llamada al sistema `unlink` elimina un nombre (enlace duro) de un fichero. Si es el último nombre que le queda al i-nodo (esto es, si su cuenta de referencias llega a cero) y ningún proceso lo tiene abierto, el fichero se elimina y se liberan todos sus recursos.

```
int unlink(const char *pathname);
```

La llamada `rmdir` elimina el directorio indicado por su parámetro. Para poder borrarlo, el directorio tiene que estar vacío. En caso de error retorna -1.

```
int rmdir(const char *pathname);
```

4.5.7. Entrada/salida con buffering

A veces, no es fácil leer de un fichero. Por ejemplo, si queremos leer un fichero línea a línea usando *read(2)*, tenemos que leer en repetidas ocasiones hasta encontrar un carácter '\n', guardando lo leído y teniendo en cuenta los tamaños de los buffers, guardar la parte de la siguiente línea, etc.

En estos casos, el algoritmo más sencillo para realizar una tarea consiste en ir leyendo carácter a carácter de un fichero. Realizar una llamada al sistema sale caro y si el fichero tiene millones de caracteres, es extremadamente ineficiente leer carácter a carácter.

Por ese motivo se suelen usar bibliotecas de entrada/salida con *buffering*. En Unix, las funciones de la biblioteca estándar de C *stdio(3)* ofrecen este tipo de operaciones con *buffering*. Llevamos ya tiempo usando funciones de stdio, la función `printf` es una función de esta biblioteca.

Una biblioteca de este tipo gestiona internamente un buffer en área de usuario y realiza llamadas al sistema `read` y `write` según va necesitando traer más caracteres (de forma similar a como se manejan las existencias de una tienda).

Por ejemplo, si lo que queremos desde nuestro programa es leer byte a byte, podemos estar llamando a esa biblioteca continuamente para leer. Internamente, no realiza una llamada al sistema para leer cada vez que le pedimos un byte. Lo que hace es rellenar su

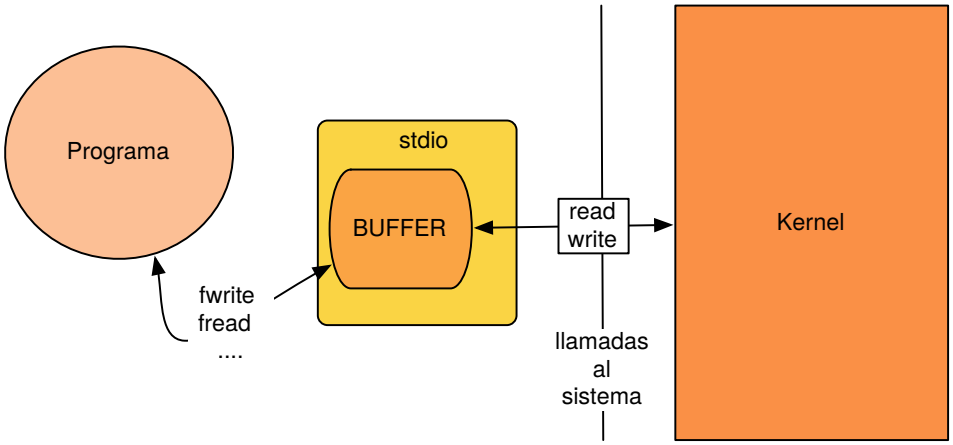

Figura 4.9: Stdio gestiona el buffer interno y sirve las peticiones desde el mismo.

buffer interno con llamadas a `read`, según lo va necesitando, e ir respondiendo a nuestras lecturas byte a byte con lo que tiene en el buffer. De esta forma es eficiente, no pierde el tiempo realizando un número grande llamadas al sistema `read` (hace menos que traen más bytes, amortizando el coste).

Stdio usa *streams*, representados por el tipo de datos `FILE`, para representar un stream abierto. De serie, nos ofrece streams apuntadas por variables globales para la entrada estándar (`stdin`), salida (`stdout`) y salida de errores (`stderr`).

Si queremos abrir un fichero para usarlo a través de stdio, tenemos la función de biblioteca `fopen`:

```
FILE * fopen(const char *restrict path, const char *restrict mode);
```

La función *fopen(3)* retorna un puntero al stream, NULL en caso de error. El primer parámetro es la ruta al fichero. El segundo es una string que describe el modo de apertura. Por ejemplo “r” para lecura, “rw” para lectura-escritura, etc.

Aunque en el estándar de C[23] existe la definición de streams de texto y streams binarios, en Unix (y, por tanto, en Linux) no hay diferencia (ver *fopen(3)*): todos los ficheros se comportan como binarios y por tanto, todos los streams son binarios. En otras palabras, si la cadena contiene “b”, se ignora. En otros sistemas puede no ser así, consulta la página de manual en tu sistema.

Si ya tenemos abierto el fichero en un descriptor de fichero y lo que queremos es usar un stream para hacer entrada/salida de ese mismo descriptor, pero con buffering, tenemos otra función. La función *fdopen(3)* configura un stream a partir de un fichero abierto. El primer parámetro es el descriptor de ficheros en el que tenemos abierto el fichero en lugar de la ruta.

[23] Concretamente en el estándar C89.

```
FILE * fdopen(int fildes, const char *mode);
```

Para cerrar un stream tenemos que usar *fclose(3)*. Esta función vacía el stream si era de escritura, descargando al fichero las partes del buffer interno que no se habían escrito, libera los recursos asociados y cierra el descriptor de fichero subyacente[24].

```
int fclose(FILE *stream);
```

La función *fread(3)* lee del stream un número de elementos (`nitems`) de tamaño `size`, los guarda en memoria a partir de la dirección `ptr`. Está pensada para trabajar leyendo datos de un tipo dado (con un tamaño dado). Retorna el número de elementos leídos.

```
size_t fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);
```

Retorna menos elementos que los solicitados (o cero) cuando llega a fin de fichero, pero también cuando tiene un error. Para diferenciar entre esos dos casos, hay que usar las funciones *feof(3)* y/o *ferror(3)*. Tanto esas funciones como *fread(3)* **no actualizan** la variable `errno`.

La función *fwrite(3)* escribe en el fichero un número de elementos (`nitems`) de tamaño `size`, de la dirección `ptr`. Retorna el número de elementos escritos.

```
size_t fwrite(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);
```

En ciertas ocasiones, necesitaremos forzar que se escriba en el fichero correspondiente lo que hemos escrito con `fwrite` y se ha quedado en buffer (esto es, todavía no ha ido al fichero). La función `fflush` fuerza el vaciado del buffer.

```
int fflush(FILE *stream);
```

[24] Ojo, si se ha utilizado *fdopen*, es peligroso llamar a *close* del descriptor directamente: puedo perder datos si se han escrito y están pendientes en el buffer. Lo mejor es olvidarse del descriptor y cerrar directamente el stream con *fclose*.

Ten en cuenta que usar un stream con buffering para trazar un programa puede causar problemas, porque seguramente no veamos la escritura cuando se realiza el `fwrite`. La vemos más tarde cuando se descarga el buffer (o nunca si el programa muere por un error). Es lo que pasa con `printf`.

Veamos el siguiente programa:

Programa 4.7: trace.c

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      int x = 13;
9
10     printf("x: %d", x);
11     for (;;) {
12         ;
13     }
14     exit(EXIT_SUCCESS);
15 }
```

Si lo ejecutamos:

```
$ ./trace
```

El programa entra en el bucle infinito sin escribir en su salida la traza. La llamada a `printf` ha escrito en el buffer intermedio de stdio y los datos siguen allí, no se han volcado al fichero (que en este caso es el terminal).

Por omisión, el stream `stderr` no tiene buffering, por eso siempre que queramos trazar un programa es mejor hacerlo a través de la salida de errores con `fprintf`. Esa función es igual que `printf`, pero escribe en el stream indicado (`printf` siempre lo hace en `stdout`).

```
int fprintf(FILE *stream, const char *format, ...);
```

Si cambiamos el programa anterior para que use `fprintf` con la salida de errores, veremos la traza antes de entrar al bucle infinito.

En realidad, stdio tiene tres tipos de buffering:

- Sin buffer (como `stderr`).
- De bloque: hasta que no se completa un bloque, no se escriben los datos al fichero o se retorna de la lectura. A esto se le llama *fully buffered*. Ese el modo que se suele usar para ficheros convencionales.

- De línea: cuando se acaba una línea, se escribe en el fichero o se retorna la de la lectura. Así es como se configura `stdin` cuando es un terminal. Si en el programa anterior terminamos la traza con un carácter de nueva línea, veremos que se imprime antes de entrar en el bucle infinito.

En general, un stream es *fully buffered* si se configura sobre un fichero convencional y de línea si se abre para un terminal [19, Capítulo 5.5]. Se puede usar la función *setvbuf(3)* para cambiar el tipo de buffering de un stream. Si además queremos especificar el tamaño del buffer intermedio, podemos usar *setbuffer(3)*.

Veamos otro ejemplo. El siguiente programa lee completo el fichero que se le pasa como argumento, byte a byte. Si se le pasa el modificador `-b` se usará buffering y si se `-n` no se usará:

Programa 4.8: buffering.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <err.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10
11 int
12 nobuffering(int fd)
13 {
14     int nr;
15     char c;
16     int count = 0;
17
18     /*
19      * Wrong and slow. Just an example.
20      */
21     while ((nr = read(fd, &c, 1)) != 0) {
22         if (nr < 0) {
23             err(EXIT_FAILURE, "cant read");
24         }
25         count++;
26     }
27     return count;
28 }
29
30 int
31 buffering(int fd)
32 {
33     char c;
34     int count = 0;
35     FILE *f;
36
37     f = fdopen(fd, "r");
38     if (f == NULL) {
39         err(EXIT_FAILURE, "cant fopen fd");
40     }
41     while (fread(&c, 1, 1, f) != 0) {
42         count++;
43     }
44     if (!feof(f)) {
45         errx(EXIT_FAILURE, "fread failed");
46     }
47     fclose(f);
48     return count;
49 }
50
51 int
52 invalidflag(char *s)
53 {
54     return strcmp(s, "-b") != 0 && strcmp(s, "-n") != 0;
55 }
```

Programa 4.9: buffering.c (cont)

```
57 int
58 main(int argc, char *argv[])
59 {
60     int fd;
61     int n;
62
63     if (argc != 3 || invalidflag(argv[1]))
64         errx(EXIT_FAILURE, "usage: %s [-b | -n] file", argv[0]);
65     fd = open(argv[2], O_RDONLY);
66     if (fd < 0) {
67         err(EXIT_FAILURE, "cannot open file");
68     }
69     if (strcmp(argv[1], "-b") == 0) {
70         n = buffering(fd);
71     } else {
72         n = nobuffering(fd);
73     }
74     printf("total: %d chars\n", n);
75     close(fd);
76     exit(EXIT_SUCCESS);
77 }
```

Si probamos el programa, ejecutando con `time` para ver cuánto tarda en ejecutar en ambos casos:

```
 $ time ./buffering  -b quijote.txt
total: 2226045 chars

real    0m0.033s
user    0m0.033s
sys     0m0.000s

 $
 $ time ./buffering  -n quijote.txt
total: 2226045 chars

real    0m0.424s
user    0m0.076s
sys     0m0.348s
```

Podemos hacer lo mismo sacando el fichero de las *cachés* del sistema para ver que en realidad, el efecto es realmente de la entrada salida con buffering y no de las cachés del sistema.

```
 $ sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'
 $ time ./buffering  -n quijote.txt
total: 2226045 chars

real    0m0.409s
user    0m0.064s
sys     0m0.344s
 $ sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'
 $ time ./buffering  -b quijote.txt
total: 2226045 chars

real    0m0.046s
user    0m0.034s
sys     0m0.004s
 $
```

Vemos que el programa que utiliza buffering es un orden de magnitud más rápido que el otro en cualquier caso.

La función *fgets(3)* lee una línea de hasta `size` bytes (independientemente del tipo de buffering que tenga el stream) y la guarda en la string `str`. Siempre deja una string terminada en un carácter nulo (por tanto la línea que se deja en el buffer sera del tamaño `size` -1). Si llega al final del fichero o hay algún error, retorna NULL. Para diferenciar entre estos casos hay que usar las funciones `feof` o `ferror` como hemos visto antes.

```
char * fgets(char * restrict str, int size, FILE * restrict stream);
```

Este programa lee todas las líneas de su entrada y las escribe en su salida:

Programa 4.10: readlines.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <err.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10
11 enum {
12     Maxline = 256,
13 };
14
15 int
16 main(int argc, char *argv[])
17 {
18     char line[Maxline];
19
20     while (fgets(line, Maxline, stdin) != NULL) {
21         printf("line: %s\n", line);
22     }
23     if (!feof(stdin)) {
24         errx(EXIT_FAILURE, "eof not reached");
25     }
26     exit(EXIT_SUCCESS);
27 }
```

Ejecutemos este programa de tres formas:

(i) Leyendo del terminal hasta que se provoca fin de fichero pulsando `Ctrl+d` (las líneas naranjas son las que se escriben en el terminal con el teclado y las negras son las que imprime el programa).

(ii) Leyendo de un pipe.

(iii) Leyendo de un fichero que provoca un error de entrada/salida para comprobar si ha salido del bucle por el final del fichero o por un error.

```
 $ ./readlines
hey
line: hey
ho
line: ho
let's go
line: let's go
^D
 $ seq 1 5 | ./readlines
line: 1
line: 2
line: 3
line: 4
line: 5
 $ ./readlines < /proc/2090/mem
readlines: eof not reached
 $
```

4.6. Ejercicios no resueltos

1. Averigua qué discos utiliza tú sistema. Recuerda que puedes usar `lsblk`, `mount`, `df` y mirar `/dev`. Mira qué particiones hay y de qué tipo son. Puedes usar `fdisk`, `parted` o `gparted`. Intenta ver cómo te representan estos comandos las particiones y compara unas con otras. Ojo, no cambies la estructura de tu sistema por accidente. Si tienes dudas, pruébalo en una máquina virtual. †∗

2. Particiona y formatea un pendrive y una imagen de disco con Ext4, móntala, crea ficheros y directorios, desmóntala.†∗

3. Formatea una imagen de disco con VFAT, móntala, crea ficheros y directorios. Desmóntala. Intenta inspeccionar su contenido imprimiendo sus bytes con `od` o `xxd` y `strings`. Identifica las estructuras de datos del sistema de ficheros (tabla FAT, entradas de directorio, etc.) † † † ∗ ∗

4. Formatea una imagen de disco con Ext4, móntala, crea ficheros, desmóntala. Intenta inspeccionar sus bytes con `od` o `xxd` y `strings`. Identifica las estructuras de datos del sistema de ficheros (superbloque, tabla de inodos, inodos, entradas de directorio, etc.). † † † ∗ ∗

5. Escribe en C una versión sencilla de `cp` que reciba sólo dos argumentos (nombre de fichero origen y destino). Que compruebe que el fichero destino no existe y en tal caso lo cree. Prueba a medir su rendimiento copiando un fichero de $1GB$ con diferentes tamaños de *buffer*: de 1 byte en 1 byte, 2, 4 y así hasta $64KB$. † ∗ ∗

6. Escribe en C una versión sencilla de `du` que escriba el tamaño y el nombre de todos los directorios y subdirectorios del directorio actual. † ∗ ∗

7. Continúa la minishell del Capítulo 3 utilizando entrada/salida con buffering. Es un bucle infinito que escribe el *prompt* por la salida de error, lee una línea con un comando sencillo con argumentos, espera a que acabe y comienza de nuevo. † † ∗∗

8. Añádele la minishell redirecciones y la posibilidad de ejecutar en background (mediante &). † † ∗∗

9. Ejecuta la minishell bajo el control de `valgrind` $--track-fds=yes$. †∗

10. Añádele variables de entorno a la minishell. † † ∗

11. Escribe un programa en C que liste el directorio raíz de un fichero con una imagen de un sistema de ficheros tipo VFAT que recibe como parámetro. † † †∗ ∗ ∗

12. Escribe un programa en C que liste el directorio raíz de un fichero con una imagen de un sistema de ficheros tipo EXT4 que recibe como parámetro. † † †∗ ∗ ∗

5 Memoria

En este capítulo estudiaremos un poco más en profundidad cómo gestiona el sistema la memoria. Empezaremos con la asignación de memoria dinámica, después estudiaremos la gestión de la memoria virtual, la paginación, etc.

5.1. Modelo básico de memoria

Recordemos que la memoria de un microprocesador se comporta como bytes direccionables. Incrementar en uno la dirección, significa pasar a direccionar el siguiente byte.

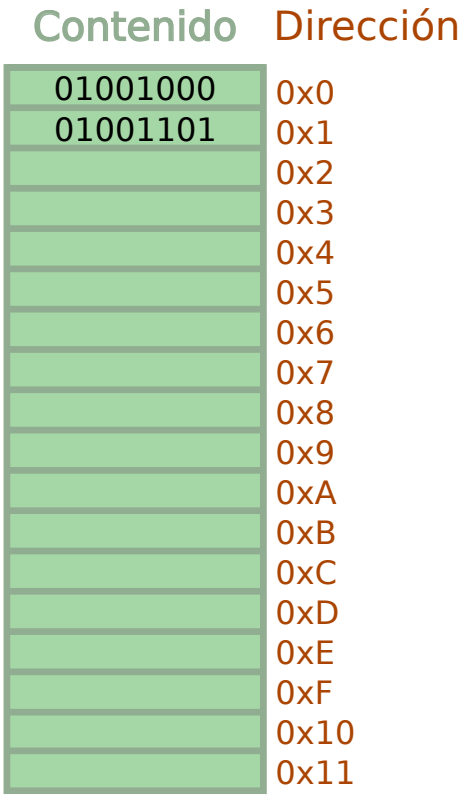

Figura 5.1: Modelo básico de memoria

Esto no quita que, debido al tamaño de los registros del procesador (que son de más de un byte, muchos hoy en día de 64 bits), haya instrucciones que trabajen con varios bytes simultáneamente. Normalmente el tamaño natural de trabajo se suele llamar palabra, aunque en una misma máquina puede haber varios tamaños *naturales* diferentes (128 bits para instrucciones de criptografía, 32 bits para coma flotante y 64 bits para instrucciones normales e incluso bloques de memoria más grandes para instrucciones vectoriales). Todo esto, sumado a los tamaños de las entradas en las cachés hardware, suele añadir restricciones de *alineamiento* al uso de la memoria. Esto significa que se suele intentar que el comienzo de una región de memoria coincida con un múltiplo de un tamaño que suele ser un superconjunto de los anteriores (64 o 128 bits normalmente o incluso

más para alinear a páginas, que veremos a continuación). Incluso hay arquitecturas que prohíben determinadas instrucciones si la memoria no está alineada. Sin embargo, no hay que olvidar que la memoria direcciona bytes.

5.2. Asignación dinámica de memoria: el heap

El problema es el siguiente: en tiempo de ejecución, un programa va demandando trozos de memoria según los vaya necesitando y los irá liberando cuando ya no los necesite. Los trozos pueden ser de diferentes tamaños y, una vez que se compromete una reserva, no se puede mover a otra parte de la memoria.

Es importante recordar y entender bien por qué el programa reserva memoria y la libera. En el modelo de proceso que hay en un sistema operativo moderno, el programa se puede considerar que vive en una máquina separada del resto (luego veremos los detalles que permiten estas suposiciones). Sin embargo, un programa aislado sigue teniendo que reservar y liberar memoria. ¿Por qué es así? La razón es que las demandas de memoria de las diferentes partes del programa (en particular, sus estructuras de datos) *compiten unas con otras por la memoria*, que es un recurso limitado.

Imaginemos un programa reproductor de video, cuya labor es leer un fragmento de la película, descomprimirlo y representar la imagen correspondiente por pantalla. La parte que descomprime el fotograma compite con la interfaz de usuario por conseguir memoria. El reproductor podría intentar reservar los 30 GB de memoria que puede ocupar la película descomprimida, pero claramente se quedaría sin memoria. Además esa parte del programa compite con las demás partes. Por tanto, el programa reserva memoria suficiente para un fotograma, lee la parte correspondiente, la descomprime, la representa y libera esa memoria[1]. De esta forma, las estructuras de datos del programa van creciendo y decreciendo para que en un instante de tiempo la memoria que requiere el programa sea razonable. Todo esto sucede para cualquier programa, incluido el sistema operativo.

Nos vamos a centrar ahora en cómo se asigna la memoria de forma dinámica desde el punto de vista de un proceso de área de usuario (aunque este problema se tiene tanto dentro como fuera del núcleo).

Ciertos lenguajes de programación tienen gestión **implícita** de la memoria. Por ejemplo, cuando reservamos memoria en Java (i.e. cuando instanciamos un objeto), después no nos tenemos que preocupar de liberarla. Estos sistemas tienen un **recolector de basura** (*garbage collector*) que se encarga de llevar la cuenta de las referencias a cada trozo reservado para liberarlos automáticamente cuando ya no están en uso. En C y otros lenguajes, la gestión de memoria es **explícita**: el programador indica cuándo ya no se necesita una reserva y se debe liberar. Para ello tiene dos funciones de biblioteca, `malloc` y `free`, que reservan y liberan memoria. Aunque se puede pedir más espacio al sistema operativo, (con `brk` como veremos luego) en una primera aproximación, se puede considerar que estas funciones de biblioteca actúan como el *maître* de un restaurante,

[1] Por supuesto, esto es una simplificación, el proceso es más complicado y hay audio, no todos los fotogramas son iguales, etc.

asignando y liberando mesas a los clientes. Aquí los clientes son las estructuras de datos del programa y las reservas (que se realizan mediante las funciones de las bibliotecas) se corresponden con regiones de memoria del proceso. El *maître* se encargará de anotar en su libro qué mesa corresponde a cada cliente, pero es responsabilidad de los clientes no tomarse la sopa de la mesa vecina, que aquí se correspondería con escribir o leer zonas de memoria que no te corresponden.

Las funciones de biblioteca correspondiente (`malloc` y `free`) reparten la memoria **sin la intervención del sistema operativo** (salvo las llamadas esporádicas a `brk` cuando se agota el segmento) totalmente en espacio de usuario. El sistema asigna un trozo grande de memoria al proceso y estas funciones lo administran. Esto se puede ver en la figura5.2.

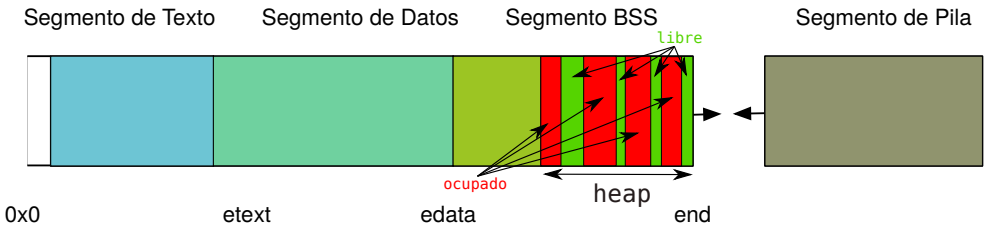

Figura 5.2: Malloc y free reparten el *heap*, de un proceso parte del `BSS`.

En el lenguaje C, el programa puede llamar a `malloc` en cualquier momento, indicando el tamaño del trozo de memoria deseado (expresado en número de bytes). La función `malloc` devuelve la dirección de memoria donde comienza el fragmento de memoria que se ha reservado. A partir de esa dirección de memoria, es seguro que tenemos disponibles el número de bytes indicado[2]. Puede que el trozo reservado sea, en realidad, más grande que lo que hemos solicitado, pero **el programador nunca puede suponer que es así**. En caso de no poder satisfacer la reserva, `malloc` devuelve un puntero nulo. Siempre debemos comprobar lo que nos devuelve `malloc`:

Programa de C 5.1: malloc

```
1 ...
2 p = malloc(sizeof(MyType));
3 if(p == NULL){
4     // handle error
5     ...
6 }
7 // use p
8 ...
9 free(p);
```

Ten en cuenta que si `malloc` falla, la situación es bastante crítica. Ya hablaremos más adelante en este capítulo sobre qué pasa si el sistema se queda sin memoria, esto es, está OOM (***O**ut **O**f **M**emory*).

[2]Por ahora suponemos que es así, más adelante hablaremos de una estrategia llamada *overcommitment*.

Una vez que ya no necesito la memoria, debo liberarla para no tener *leaks* de memoria. Un leak es un trozo de memoria que no vamos a poder liberar nunca porque hemos perdido sus referencias. Si un programa pierde memoria sistemáticamente, al final acabará con toda la memoria que tiene disponible.

Existen múltiples herramientas para detectar leaks de memoria. Por ejemplo, `valgrind`[3] es un analizador dinámico que informa de este tipo de errores. Supongamos el siguiente código:

Programa 5.2: leaks.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <errno.h>
5
6  int
7  main(int argc, char *argv[])
8  {
9      int i;
10     int max;
11     char *p;
12
13     max = 0;
14     for (i = 1; i < argc; i++) {
15         if (strlen(argv[i]) > max) {
16             max = strlen(argv[i]);
17             p = malloc(max + 1);     // leak!
18             strcpy(p, argv[i]);
19         }
20         printf("[%d] %s\n", i, argv[i]);
21     }
22     printf("The largest argument is: %s\n", p);
23     free(p);
24     exit(EXIT_SUCCESS);
25 }
```

El programa se queda con el argumento más largo para imprimirlo al final:

```
 $ ./a.out lespaul sg jaguar jazzmaster telecaster stratocaster
[0] ./a.out
[1] lespaul
[2] sg
[3] jaguar
[4] jazzmaster
[5] telecaster
[6] stratocaster
The largest argument is: stratocaster
 $
```

Este código tiene un error. Cada vez que se encuentra un argumento mayor que el que teníamos, se llama a `malloc` sin liberar el que hasta ahora era el argumento más largo. Esto es, tiene un *leak* de memoria. Si ejecuntamos el código con el analizador dinámico, nos avisará de ello:

[3] Puedes instalarlo con `apt install valgrind`.

```
 $ valgrind ./a.out lespaul sg jaguar jazzmaster telecaster stratocaster
==6491== Memcheck, a memory error detector
==6491== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6491== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6491== Command: ./a.out lespaul sg jaguar jazzmaster telecaster stratocaster
==6491==
[1] lespaul
[2] sg
[3] jaguar
[4] jazzmaster
[5] telecaster
[6] stratocaster
The largest argument is: stratocaster
==6491==
==6491== HEAP SUMMARY:
==6491==     in use at exit: 19 bytes in 2 blocks
==6491==   total heap usage: 4 allocs, 2 frees, 1,056 bytes allocated
==6491==
==6491== LEAK SUMMARY:
==6491==    definitely lost: 19 bytes in 2 blocks
==6491==    indirectly lost: 0 bytes in 0 blocks
==6491==      possibly lost: 0 bytes in 0 blocks
==6491==    still reachable: 0 bytes in 0 blocks
==6491==         suppressed: 0 bytes in 0 blocks
==6491== Rerun with --leak-check=full to see details of leaked memory
==6491==
==6491== For lists of detected and suppressed errors, rerun with: -s
==6491== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
 $
```

Como se puede observar, en el resumen se indica que se han perdido definitivamente dos bloques. Esta herramienta es muy útil para depurar errores relacionados con la memoria[4].

Como ya sabemos, la región de la memoria de un proceso que está destinada a la reserva de memoria dinámica es el ***heap*** o *montón*, normalmente situada al final del segmento `BSS`. El heap mantiene una estructura interna para marcar las regiones que están en uso y las que están libres. Ten en cuenta que todas las partes de esa región son direcciones *válidas* (esto es, si intentamos acceder a ellas no tendremos un fallo fatal, como un *segmentation fault* o *bus error*), pero pueden estar marcadas como usadas o libres de cara a nuestro programa.

5.2.1. Mecanismos

Un heap se suele implementar como una o varias listas enlazadas[5]. Las listas puede ser de trozos libres, de trozos ocupados o de trozos libres y ocupados. La lista también puede ser simple o doblemente enlazada, circular, etc. Además, puede usar distintas listas. Siguiendo con la analogía, los punteros al siguiente y demás de estas listas enlazadas son como el libro donde el *maître* anota qué mesas están reservadas. Por supuesto también

[4] Observa que la propia herramienta te indica que cómo ejecutarlo si queremos más detalles de los errores (en este caso, con el parámetro `--leak-check=full`).

[5] Técnicamente, los más avanzados son árboles.

viven en memoria y competirán con la memoria a reservar[6].

Cuando se libera un trozo, que se corresponde con un nodo en la lista enlazada, y resulta que algún trozo adyacente (es decir, un nodo que está justo antes o después en memoria, o ambos) está libre, podemos fundir todo en un único nodo representando ese trozo libre (*coalescing*). La política dictará cuándo debemos fundir dos trozos adyacentes. No fusionar puede limitar a la hora de reservar trozos más grandes, pero fusionar también tiene un coste en tiempo.

Cada trozo (nodo) enlazado puede tener cabeceras (*headers*) y pies (*footers*) con los metadatos necesarios para mantener la estructura del heap y moverse rápido por ella. Cuando se usan los dos, se puede acelerar el fundido. Naturalmente, la memoria consumida por pies y cabeceras no puede ser utilizada por el programa cliente, por tanto hay un coste en espacio, como hemos dicho antes.

Por ejemplo, en la figura 5.3 se muestra un ejemplo con un heap implementado con una lista doblemente enlazada de trozos libres, con cabeceras y pies. Como se puede observar, los dos últimos trozos están libres y se podrían fundir en un único trozo libre.

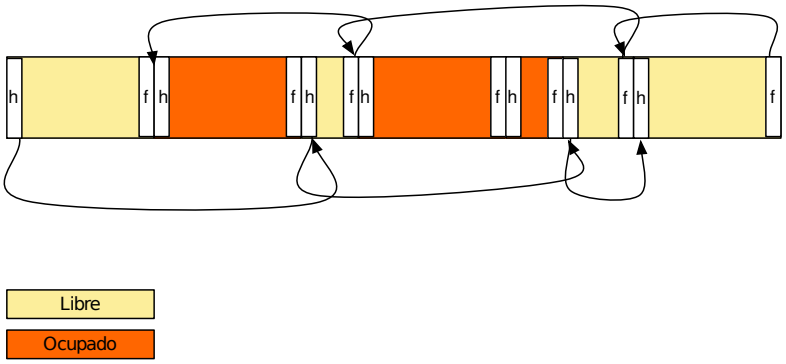

Figura 5.3: Ejemplo de heap implementado con una lista doblemente enlazada de trozos libres.

Muchas veces nos interesa *envenenar* las zonas liberadas para descubrir errores en los programas. El envenenamiento consiste en escribir esas zonas con un valor conocido y fácilmente identificable en hexadecimal (p. ej. `0xdeadbeef` o `0xcafebabe`). Si desde un programa vemos que estamos usando memoria con esos valores, podremos detectar rápidamente que estamos usando memoria liberada, etc. Además, en muchos casos el programa cliente fallará por intentar utilizar datos inválidos (por ejemplo punteros). Esto, que parece peor, en realidad es bueno. Es mejor que un programa falle estrepitosamente que se quede silenciosamente corrompiendo memoria sin dar indicaciones de qué está sucediendo.

Cuando ya no nos quedan trozos libres en el *heap*, se puede intentar hacer crecer la región de memoria para conseguir más espacio libre para repartir pidiéndosela al sistema operativo. En Unix, las llamadas `brk` y `sbrk` sirven para esto. La implementación de

[6] El *maître* tendrá su mesa con el libro de reservas, aunque quizás esto es llevar ya demasiado lejos la analogía

`malloc` se apoya en ellas, los programas de usuario no usan `brk` o `sbrk` directamente (ver *brk(2)*).

Muchos de los errores de programación relacionados con la reserva dinámica de memoria consisten en desbordamientos. Esto pasa cuando escribimos más datos que los que nos entran en el trozo que nos han dado. Cuando pasa esto, se puede corromper la estructura de datos interna del asignador (los pies y las cabeceras, etc.).

Si esto ocurre, la lista enlazada es inconsistente: a la hora de recorrerla podemos tener múltiples tipos de fallos. Una forma de evitar este tipo de errores es probar nuestros programas con una analizador dinámico que compruebe las estructuras de datos internas del heap. El analizador dinámico `valgrind`, del que ya hemos hablado antes, informa también de fallos que provocan la corrupción del heap (y de muchas otras cosas).

5.2.2. Políticas

Las estructuras de datos de las que hemos hablado son *mecanismos*. Son las herramientas que nos dan la posibilidad de implementar diferentes estrategias dependiendo de los compromisos de diseño que queramos tomar[7]. Estos compromisos los implementan algoritmos que hacen uso de los mecanismos (son dos caras de una misma moneda) y que reciben el nombre de *políticas*. Es útil en el diseño de un programa, en este caso una biblioteca de asignación, separar claramente políticas de mecanismos y hacer, que en la medida de lo posible, las implementaciones sean independientes. Así, se dejarán claros los compromisos de diseño y además será más fácil adaptar el programa en caso de ser necesario.

Para repartir el heap podemos aplicar distintas políticas. Antes de seguir, necesitamos refrescar los dos tipos de fragmentación, que van a ser costes de memoria de nuestros compromisos de diseño.

La **fragmentación externa** ocurre cuando después de reservar/liberar en repetidas ocasiones trozos de distinto tamaño, quedan huecos de un tamaño tan pequeño que en general resultarán inservibles. Sin embargo, la suma de esos fragmentos libres sí sería útil. Se puede usar esta analogía para describir su efecto: es lo que pasa cuando vamos 4 personas al cine y quedan 12 butacas libres, pero no hay 4 contiguas.

La ley del 50% dice que dados N bloques, $0.5 * N$ se pierden por la fragmentación externa (en media), ver el Capítulo 7 de [2][8].

La *compactación* (la reagrupación de las reservas para que sean contiguas) solucionaría el problema, pero como ya hemos explicado antes, una vez comprometida una reserva de memoria no se puede mover (¿qué haríamos con los punteros ya asignados?).

Una solución parcial a la fragmentación externa es reservar en base a trozos de tamaños fijos, de tal forma que un hueco libre siempre puede ser útil. Además, haciendo esto no malgastamos recursos para apuntar huecos inservibles. A cambio, aparece la

[7] Un ejemplo de extremo en el compromiso es el `bump allocator`. Consta de un sólo puntero. Básicamente reservar una cantidad de memoria es incrementar el puntero. Nunca se libera. Esto funciona bien en un periodo corto, por ejemplo, en el proceso de arranque de un sistema, pero claramente, si el sistema ejecuta durante suficiente tiempo, se queda sin memoria.

[8] Esto es así con una política First Fit.

fragmentación interna. Se puede usar esta analogía para describirla: es lo que pasa si vamos 2 personas a cenar y nos dan una mesa de 4 comensales. Sobra espacio dentro del trozo reservado, y la suma del espacio sobrante en todas las reservas podría ser útil.

Hay distintas estrategias, pero como siempre, no podemos querer todo a la vez y los compromisos van a depender del uso concreto que se vaya a hacer:

- Para minimizar fragmentación externa: establecer un tamaño mínimo para las reservas (nunca se rompe un trozo de tamaño mínimo), agrupación de reservas relacionadas (tiempo y tamaño), etc.
- Para minimizar fragmentación interna: buscar el mejor ajuste, etc.
- Para minimizar tiempo en las reservas: estructuras de datos más sofisticadas, uso de cachés, segregación por tamaño, etc.

A continuación, al describir las políticas, daremos algunas nociones sobre su rendimiento en general (en base a la bibliografía), pero hay que tener en cuenta que hasta que no midamos en una máquina concreta y para una aplicación específica, es difícil decir qué política se comporta mejor o peor. Se ha descrito en la literatura y los autores hemos visto personalmente en repetidas ocasiones que algoritmos que teóricamente se comportan bien, en cierto hardware y para algunas aplicaciones no lo hacen. También hemos visto lo contrario. Hay que tener en cuenta que todo lo que trataremos en siguientes secciones también puede tener un gran impacto en el comportamiento de estas políticas (fallos de página, alineamiento, caché hardware de distintos niveles, etc.). Por ahora, ignoraremos esos factores.

La política ***First Fit*** es muy sencilla, es la forma directa de implementar un heap. Simplemente se busca el primer trozo libre en el que entra la reserva, empezando por el principio. El trozo elegido se parte en dos (uno del tamaño requerido y otro del sobrante, que quedará libre). El algoritmo es simple y rápido, porque busca lo menos posible (no tiene que inspeccionar la lista completa). El problema es que puede provocar fragmentación externa. Esa fragmentación provoca pérdida de memoria y obliga a inspeccionar trozos que no sirven porque son muy pequeños (ralentiza la búsqueda).

Next Fit es parecido: se elige el primer trozo en el que cabe, empezando a buscar por donde se quedó la última vez. Empíricamente se comporta peor que First-Fit, provoca más fragmentación [2].

Best Fit selecciona el trozo del heap que se ajuste mejor al tamaño solicitado. Ayuda a tener los fragmentos pequeños cuando hay fragmentación. Es más lento que los anteriores porque tiene que inspeccionar todos los trozos libres antes de elegir uno. También tiende a dejar huecos muy pequeños o muy grandes.

La idea contraria a la anterior es ***Worst Fit***. Esta política elige el trozo que se ajuste peor al tamaño que pedimos, esto es, el más grande. Es igual de lento que el anterior, porque se tiene que recorrer toda la lista. En general, se considera peor que las políticas anteriores.

Con **Quick Fit** se mantienen listas de de trozos de tamaños populares para hacer reservas rápidas. La idea es segregar: tener distintas listas (o *memory pools*) para trozos

de distintos tamaños populares. Cuando nos piden un reserva que encaja con un tamaño popular, directamente se da el primer trozo de la lista correspondiente. Si no nos piden un tamaño popular, aplicamos otra política de las anteriores en otra lista general para satisfacer esa reserva. Esta política acelera la búsqueda y se comporta bien. Como contra, es más complejo de implementar y necesita más estructuras.

Otro algoritmo algo diferente a los anteriores es el ***Buddy System***. En este algoritmo sólo se contemplan trozos que sean potencias de dos. Cuando se solicita un trozo de memoria, se redondea la reserva al tamaño de la potencia de dos más ajustada en la que cabe (p. ej. si quieres 48 bytes, se redondea a 64 bytes). Si no hay ningún trozo libre de ese tamaño, se elige uno mayor y se parte en el menor número de fragmentos (siempre con tamaño potencia de dos) para poder conseguir uno del tamaño deseado. Cuando se libera, se fusionan todas las partes que se pueda dejando el mayor trozo potencia de dos. La idea general se entiende mejor con un ejemplo. Supongamos que tenemos una memoria de 256:

1. Inicio, todo el área está libre:

   256(libre)

2. Pido 21:

   32(ocupada) → 32(libre) → 64 (libre) → 128(libre)

3. Pido 44:

   32(ocupada) → 32(libre) → **64 (ocupada)** → 128(libre)

4. Pido 51:

   32(ocupada) → 32(libre) → 64 (ocupada) → **64(ocupada)** → 64 (libre)

5. Libero el de 21 (del paso 2):

   64 (libre) → 64 (ocupada) → 64(ocupada) → 64 (libre)

Otra forma de segregar es intentando aprovechar la localidad temporal (en lugar de la localidad por tamaños), separando la memoria en generaciones y agrupando en estructuras de datos las reservas hechas cerca en el tiempo.

En un sistema moderno, la gestión del heap es más complicada, no se suele usar simplemente uno de estos algoritmos. Las implementaciones pueden mezclar distintas políticas y ser bastante complejas. Un ejemplo real de una implementación de `malloc` de área de usuario es la siguiente:

- Para peticiones de más de 512 bytes, se usa un asignador Best Fit puro.
- Para peticiones de menos de 64 bytes, se usa un asignador Quick Fit con trozos de ese tamaño.
- Para peticiones intermedias, se usa una política mezcla de las anteriores.

- Para peticiones muy grandes, de más de 128 KB, el trozo devuelto no será parte del *heap*: se crea una región nueva de memoria con la llamada al sistema *mmap(2)* (que veremos más adelante).

Además, en sistemas como GNU/Linux podemos cambiar el comportamiento de `malloc` (p. ej. cambiando el valor de variables de entorno) y así cambiar de política si es necesario.

Otro ejemplo real: el kernel de Linux usa un Buddy System para gestionar grandes zonas contiguas de memoria física. Sobre él, se monta el *slab allocator*, que es básicamente un asignador Quick Fit con distintas cachés para trozos de tamaños populares dentro del kernel. La implementación de malloc del kernel, `kmalloc`, se apoya a su vez en este slab allocator: simplemente busca la caché que mejor se ajusta al tamaño solicitado y le solicita un trozo [24, Chapter 8]. No es más que un asignador rápido de propósito específico diseñado para el tipo de presión de memoria que tiene el kernel de Linux.

Algo que se hace a veces, cuando se sabe que se va a utilizar un patrón específico de memoria (especialmente si es malo para la implementación del asignador por defecto y fragmenta mucho la memoria) es utilizar un asignador (*allocator*) específico. Por ejemplo, un programa que manda mensajes de red de tamaños pequeños pero conocidos, puede hacer un sencillo asignador específico para aliviar la presión del heap. Pide un trozo grande de memoria y lo reparte mediante este asignador. Así, evita fragmentar la memoria. Incluso, en algunas ocasiones, se puede liberar el trozo entero de golpe (imaginemos un programa que se baja algo de la red y luego pasa a ejecutar en local, por ejemplo, puede liberar todos los mensajes de red de golpe). Estas cosas añaden complejidad y hay que ser cuidadoso midiendo el uso real de memoria y la fragmentación, haciendo *profiling* del programa antes de decidirse a implementarlas.

5.2.3. Asignadores modernos y sin fragmentación

En los asignadores modernos, se considera que hay dos patrones de uso de memoria comunes que dependen del lenguaje de programación utilizado. Los lenguajes más orientados a valores y direcciones de memoria (por ejemplo C, C++ o Go) y los que utilizan principalmente referencias (Javascript, Python, Java).

Los punteros apuntan a una dirección de memoria, las referencias apuntan a una variable. Se usan de diferente forma: los punteros se atraviesan (*dereferencia o indirección*) para operar con los datos apuntados, y podemos trabajar con la dirección (en algunos lenguajes, como en C, podemos hacer *aritmética de punteros*); cuando trabajamos con referencias, siempre trabajamos con la variable apuntada, no existe el concepto de *indirección*. Al dar los punteros acceso explícito a un elemento que contiene la dirección, el programador tiene mayor control sobre cómo se organizan los valores, incluyendo dichas direcciones, en memoria. Además, los arrays y los registros contienen valores en lugar de referencias. Como consecuencia, la forma en la que estos lenguajes usan la memoria es muy diferente y los problemas de fragmentación también. Para los segundos (los que utilizan principalmente referencias), se genera mucha más fragmentación y, normalmente, los asignadores (y recolectores de basura) tienen que ser generacionales (se van abriendo nuevas regiones cada cierto tiempo) y compactar (tienen que, efectivamente, parar el

programa, mover las regiones asignadas y reescribir los punteros). Esto es muy costoso para lenguajes con aritmética de punteros (C, C++) y además innecesario. Muchos autores modernos consideran que, para los programas escritos en estos lenguajes, en gran parte (siempre habrá excepciones) se ha resuelto el problema de la fragmentación a partir de la implementación de asignadores como `tcmalloc` [25], como analizó por primera vez el artículo seminal [26]. La idea consiste en, para este tipo de lenguajes, seguir las siguientes políticas:

- Realizar el *coalescing* instantáneamente.
- Mantener las reservas ordenadas por proximidad, identificando dos políticas que funcionan bien:
  - First fit con direcciones ordenadas por proximidad de la dirección, por ejemplo, mediante un árbol cartesiano[9].
  - Best fit.
- Poner una caché por tamaños (Quick Fit) por hilo de ejecución y una caché central compartida para el caso de agotar la primera.

Con estas políticas, se identifica que la fragmentación no se acumula y deja de ser un problema para los tipos de uso indicados anteriormente.

5.3. Memoria virtual

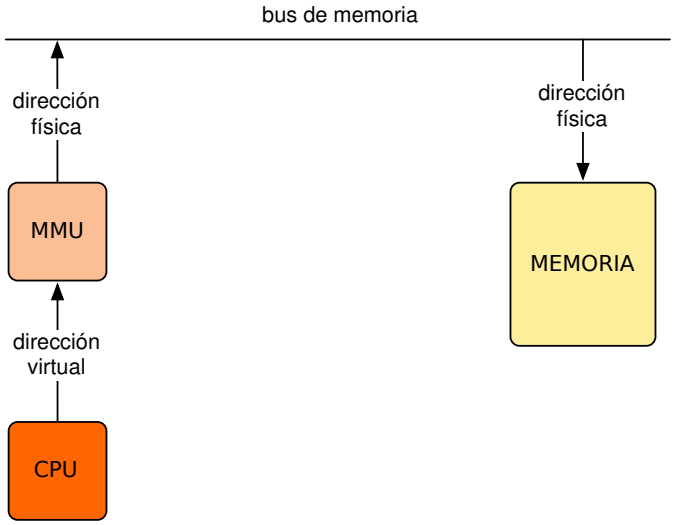

Figura 5.4: La CPU siempre usa direcciones de memoria virtual.

Lo primero que tenemos que tener en cuenta es que la CPU trata con direcciones de memoria virtual, como se muestra en la figura 5.4. La idea fundamental es la siguiente: las

[9] Un árbol binario que se deriva de una sucesión de números.

partes de la memoria de un proceso se pueden localizar en diferentes zonas no contiguas de la memoria física, como se muestra en la figura 5.5.

Una dirección virtual se traduce o *proyecta* a una dirección de memoria física usando un mapa de traducciones. Ese mapa de traducciones se llama *tabla de páginas*. En kernel mantiene para cada proceso su correspondencia o traducción de direcciones virtuales a direcciones físicas. Esto es, cada proceso tiene su tabla de páginas. Cuando se cambia el contexto a un proceso (cuando se le pone en la CPU para ejecutar), se debe configurar la CPU para que use su tabla de páginas. Un componente hardware llamado MMU (***M**emory **M**anagement **U**nit*) es el encargado de consultar la tabla, realizar las traducciones y comunicarse con la memoria a través del bus.

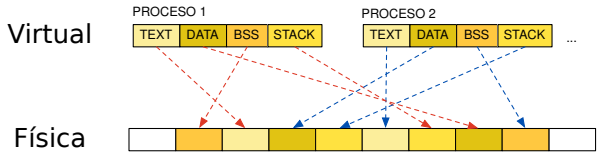

Figura 5.5: Idea principal: las distintas partes de los procesos se sitúan en diferentes partes no contiguas en la memoria física. Ten en cuenta que el *grano* para *partir* esas zonas es más pequeño que el mostrado en esta imagen (i.e. páginas).

En ocasiones, el kernel sí necesita manejar direcciones físicas. Por ejemplo, necesita comunicarse con el hardware escribiendo en direcciones físicas. En el siguiente ejemplo vemos algunas direcciones de memoria que se usan para comunicarse con los dispositivos PCI:

```
 $  cat /proc/iomem | grep PCI | head -10
000a0000-000bffff : PCI Bus 0000:00
000c0000-000c3fff : PCI Bus 0000:00
000c4000-000c7fff : PCI Bus 0000:00
000c8000-000cbfff : PCI Bus 0000:00
000cc000-000cffff : PCI Bus 0000:00
000d0000-000d3fff : PCI Bus 0000:00
000d4000-000d7fff : PCI Bus 0000:00
000d8000-000dbfff : PCI Bus 0000:00
000dc000-000dffff : PCI Bus 0000:00
000e0000-000e3fff : PCI Bus 0000:00
 $
```

Para ello, un truco es instalar un *mapa identidad*, con el que una dirección virtual se traduce a la misma dirección física.

El espacio de direcciones virtuales se suele dividir para direcciones del kernel y direcciones de usuario. En Linux es como se muestra en la Figura 5.8. En AMD64, aunque

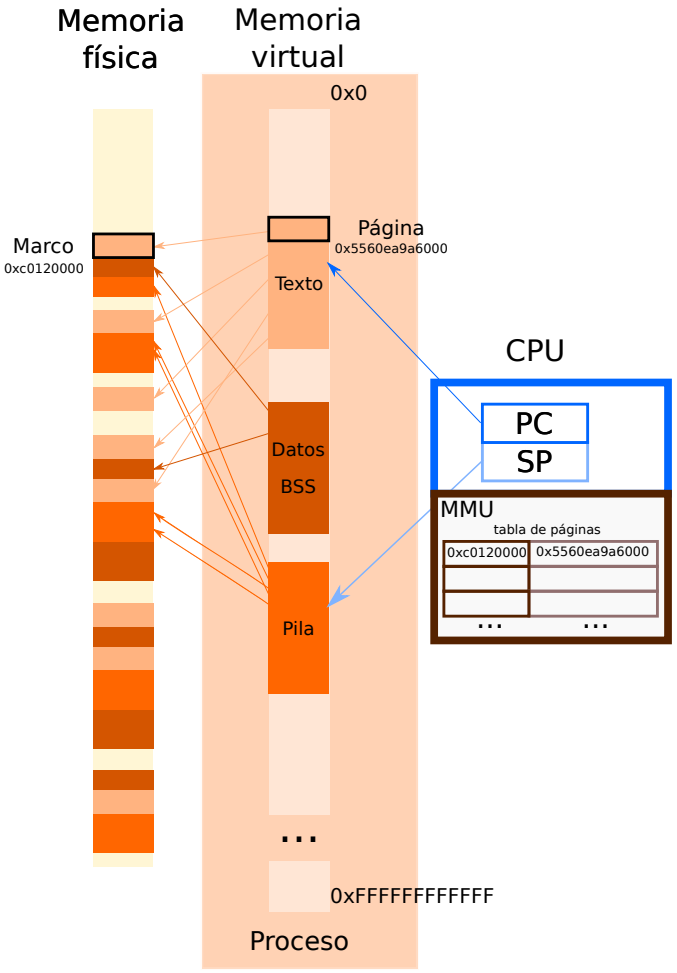

Figura 5.6: Los procesos tienen un espacio virtual que se traduce a partes no contiguas en la memoria física.

sea una máquina de 64 bits, las direcciones son de 48 bits (más que suficiente). Por tanto, cuando vemos una dirección de memoria, con un simple vistazo podemos saber si se trata de área de usuario o una dirección del memoria del kernel.

Los objetivos de la memoria virtual son varios:

- **Protección**: un proceso no puede acceder a la memoria de otro proceso ni a la del kernel si no está configurado para ello. No es posible que un proceso interfiera con otro por escribir en partes de su memoria, ya sea de forma accidental o premeditada. El sistema tiene mecanismos para compartir memoria entre procesos, pero en general, la memoria de un proceso es privada.
- **Simplicidad**: para el proceso, toda la memoria es contigua, pero en realidad no lo es (como se puede ver en las Figuras 5.6 y 5.7). Esto facilita la implementación de los programas y del desarrollo (enlazador, cargador, asignador de memoria...).

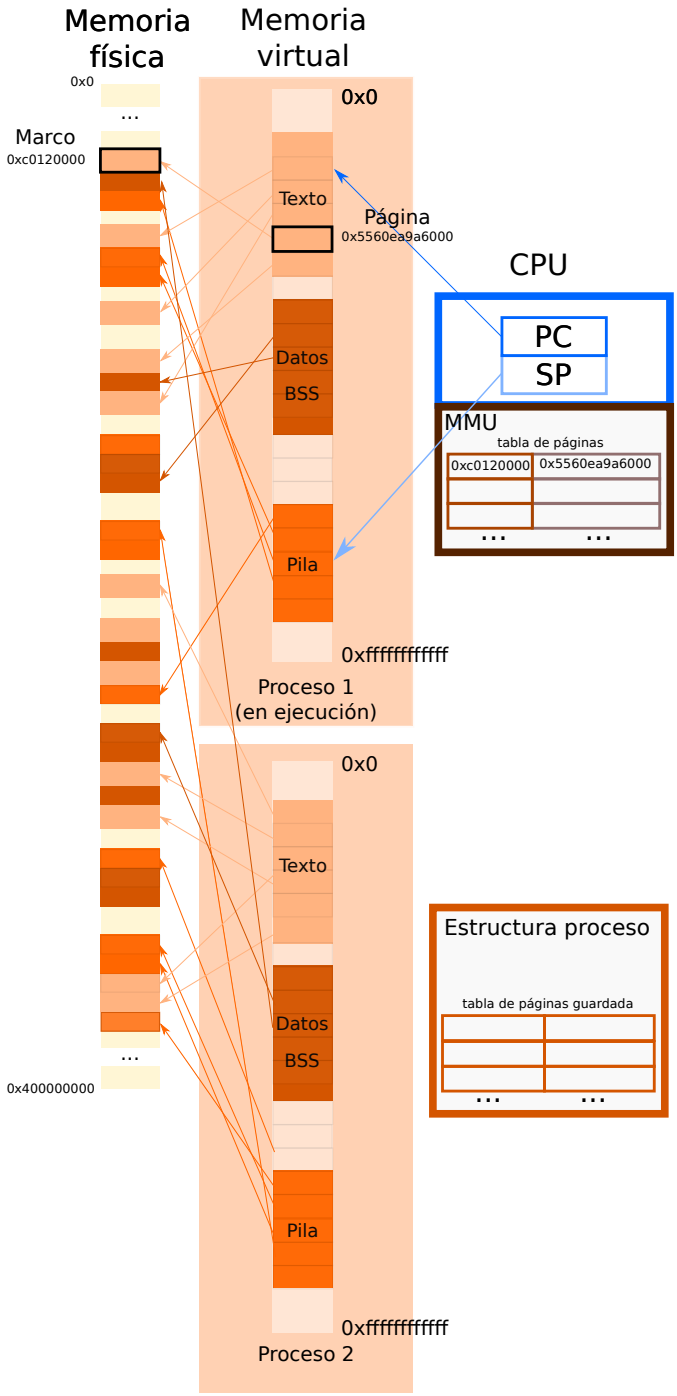

Figura 5.7: Cada proceso tiene su propio espacio de direcciones virtuales (que suele ser mucho más grande que la física).

- **Abstracción**: un proceso *cree* que está en su propia máquina y que toda la memoria es suya. De nuevo, el sistema oculta los detalles de la máquina y la concurrencia subyacente. Un proceso gestiona su memoria como si fuese el único que está ejecutando en la máquina.
- **Depuración**: la organización de la memoria es similar para todos los procesos y suele ser la misma para distintas ejecuciones de un programa[10]. Por tanto, la depuración de los programas es más fácil y los errores de programación son más reproducibles.
- **Vinculación**: permite que la resolución de símbolos y relocalizaciones se realicen en tiempo de compilación o de ejecución según sea necesario.
- **Asignación**: permite repartir la memoria RAM de forma eficiente, con poca fragmentación y en demanda.
- **Reutilización/compartición**: sigue permitiendo la compartición, ya que una zona de memoria física se puede proyectar en distintas zonas de la memoria virtual de distintos procesos, evitando la copia de regiones que son iguales para distintos procesos (p. ej. las bibliotecas).
- **Intercambio**: se puede utilizar almacenamiento secundario para almacenar la memoria de un proceso si no tenemos suficiente memoria física. Es un error pensar que esta es la única (o la principal) utilidad de la memoria virtual. En general, hoy en día este mecanismo no es tan importante como en el pasado, porque las máquinas tienen una gran cantidad de memoria RAM. Actualmente, la labor principal del intercambio es salvar el estado de la RAM en caso de suspensión o hibernación para prepararse en caso de quedarse sin batería. Normalmente en suspensión apaga prácticamente todo menos la RAM y en hibernación se copia el contenido de las páginas sucias a la zona de intercambio.

5.3.1. Páginas

La memoria física se divide en trozos de un tamaño fijo, normalmente 4 KB, pero las hay más grandes (superpáginas). El objetivo es poder tener la memoria de un proceso distribuido en zonas no contiguas de memoria física.

La terminología es la siguiente: la memoria física se divide en **marcos de página** y la memoria virtual se divide en **páginas**. Algunas partes de la memoria de un proceso (i.e. sus páginas) pueden estar en *almacenamiento secundario* (p. ej. en intercambio o en el fichero ejecutable). Cuando se rellena el contenido de una página en memoria RAM, tiene que tener un **marco de página** asignado.

La tabla de páginas del proceso tiene la dirección base para cada página. El kernel lleva la cuenta de los marcos de página que hay disponibles y según se van necesitando

[10] Hay mecanismos de seguridad que hacen que esto no sea siempre cierto.

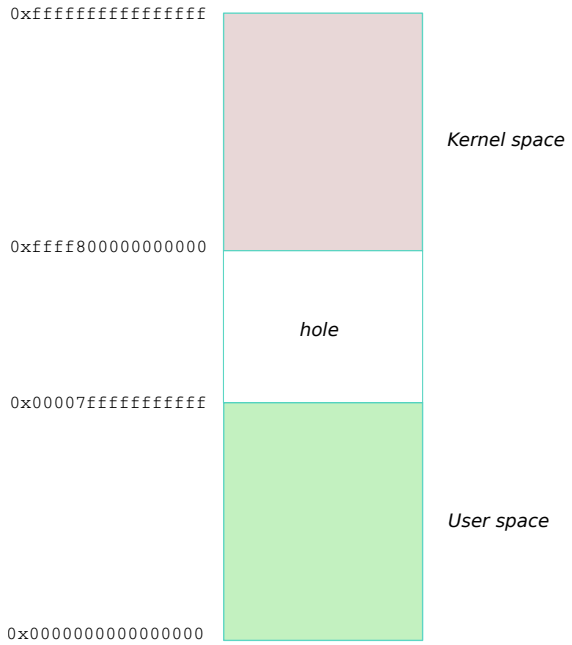

Figura 5.8: Espacios de memoria para usuario y kernel en Linux.

para alojar páginas del proceso, los va asignando y añadiendo traducciones a la tabla de páginas.

Cuando una instrucción intenta acceder a una dirección de memoria (virtual, recuerda que la CPU sólo maneja direcciones virtuales), si no hay traducción para esa página saltará un **fallo de página**: un tipo de *interrupción software*, tambien llamada *excepción*. En ese caso, el control pasa al kernel, que maneja el fallo y tendrá que ver qué ha pasado. Por ejemplo, un **fallo de segmento** (*segmentation fault*) es cuando ocurre un fallo de página y el sistema se da cuenta de que no hay traducción posible para la página en el contexto del proceso actual. Dicho de otra forma, el programa se ha salido de las regiones que tienen sentido en su espacio de memoria virtual. Más adelante veremos más sobre esto.

La figura 5.9 muestra dirección de memoria virtual. La dirección se divide en la dirección de la página (base), y el desplazamiento dentro de esa página para el byte correspondiente. El tamaño de la página será 2^m bytes (un byte es la unidad mínima de memoria direccionable).

Como vemos en la figura 5.10, cuando se accede a una dirección virtual, se usa el número de página (p) de la dirección para encontrar la dirección física del marco de página. Después se usará el desplazamiento (d) para alcanzar el byte indicado dentro de ese marco de página.

Al dividir la memoria en páginas, tendremos de nuevo fragmentación interna. Si el tamaño de página es pequeño, entonces habrá menos fragmentación, pero hay un problema: necesitamos tablas de página con muchas entradas, que implica que la tabla sea muy grande. Como consecuencia, la latencia provocada por los accesos a memoria y la

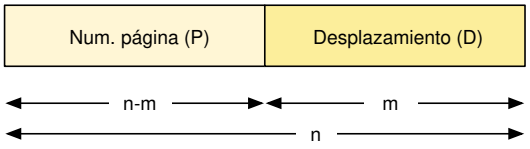

Figura 5.9: Dirección virtual.

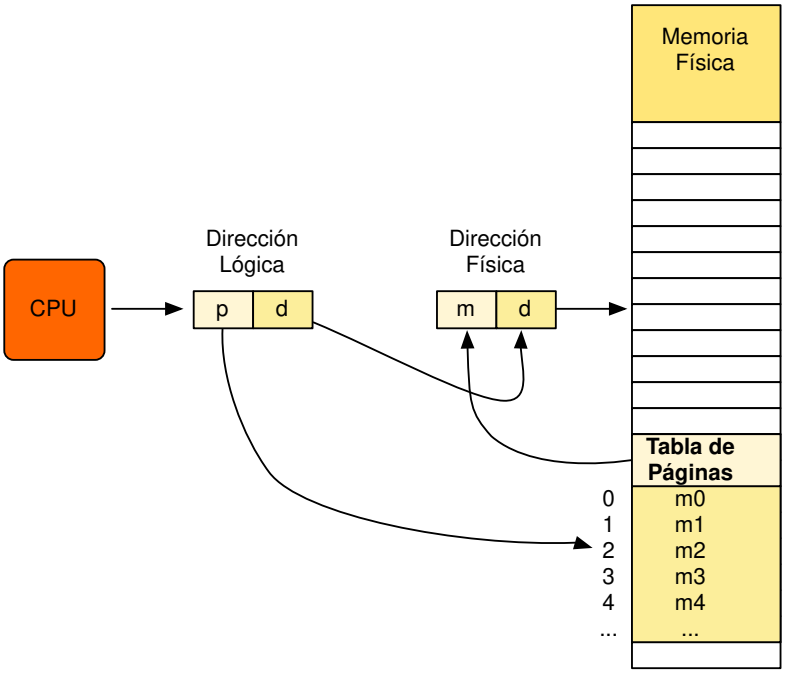

Figura 5.10: Esquema de paginación simple.

entrada/salida será muy elevada, y además las cachés tendrán que ser más grandes para ser eficaces. Por otro lado, si el tamaño de página es grande, ganaremos en tiempo de entrada/salida a la hora de traer páginas de almacenamiento y tendremos una tabla de páginas más pequeña. Normalmente el tamaño de página es de 4 KB. Algunas arquitecturas tienen, además, páginas más grandes (superpáginas de p. ej. 256 MB o incluso 2GB) normalmente en esquemas multinivel que veremos más adelante.

Las entradas de la tabla de páginas se denominan PTEs (***P**age **T**able **E**ntry*). Normalmente una PTE ocupa como una dirección de memoria (por ejemplo 64 bits). Los bits de menos peso, que se corresponden con el desplazamiento podrían estar a cero en la tabla de páginas (ver figura 5.11). Ahí normalmente se añaden bits extra con información útil, dependiendo de la arquitectura:

- Bit de presencia (o bit de válido): bit en cada PTE que indica si la página tiene

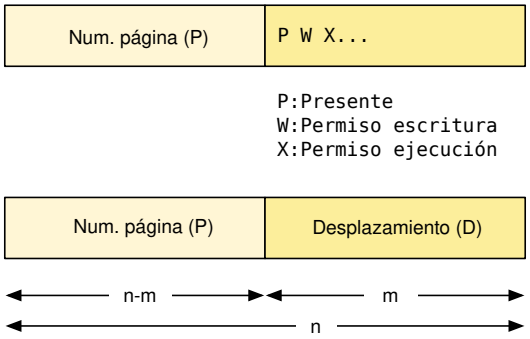

Figura 5.11: PTE.

traducción o no. Si la página no tiene traducción y el bit está a 0 puede ser porque sea no válida o porque esté en *swap*. Este bit a 1 implica que la MMU hace automáticamente la traducción al acceder a la dirección asociada a la PTE.

- Bit de modo: permiso para escribir la página o no. Si el bit está a 0 la página es de sólo lectura.
- Bit de referencia: se activa cuando se accede a la página.
- Otros bits: si puede ir a caché, permiso de ejecución, etc.

La tabla de páginas es un atributo más de un proceso, forma parte de su contexto. Para realizar un cambio de contexto se debe cambiar la tabla de páginas (se debe instalar la del proceso que va a ejecutar ahora). La tabla de páginas está en la propia memoria y se suele referenciar usando un registro especial de la CPU (los detalles dependen de la arquitectura).

Por tanto, si la tabla está en memoria y se necesita consultar para acceder a una dirección de memoria, cada acceso a la memoria por parte de un programa necesitará en realidad dos accesos de memoria: (1) el acceso a la tabla de páginas necesario para la traducción y (2) el propio acceso a la memoria que se necesitaba hacer.

TLB

El hardware moderno usa una pequeña memoria caché asociativa para ahorrar en la medida de lo posible el paso (1). La TLB (***T**ranslation **L**ook-aside **B**uffer*) es una pequeña memoria caché de la tabla de páginas (unos cientos de entradas). Es una memoria muy rápida. Para hacernos una idea, el acceso a cada uno de estos niveles de memoria es:

1. TLB, SRAM fully-associative: 1 ciclo.
2. Caché L1, SRAM set-associative: 3 ciclos.
3. Caché L2, SRAM: 14 ciclos.

4. Memoria principal, DRAM: 240 ciclos.

Cuando la MMU necesita traducir la dirección de una página, si está en la TLB, se traduce directamente. Si no está, se busca en la tabla de páginas. Por tanto, cuanto mayor tamaño de página, mayor tasa de acierto en TLB. En la figura 5.12 se muestra el esquema de traducción cuando tenemos una TLB.

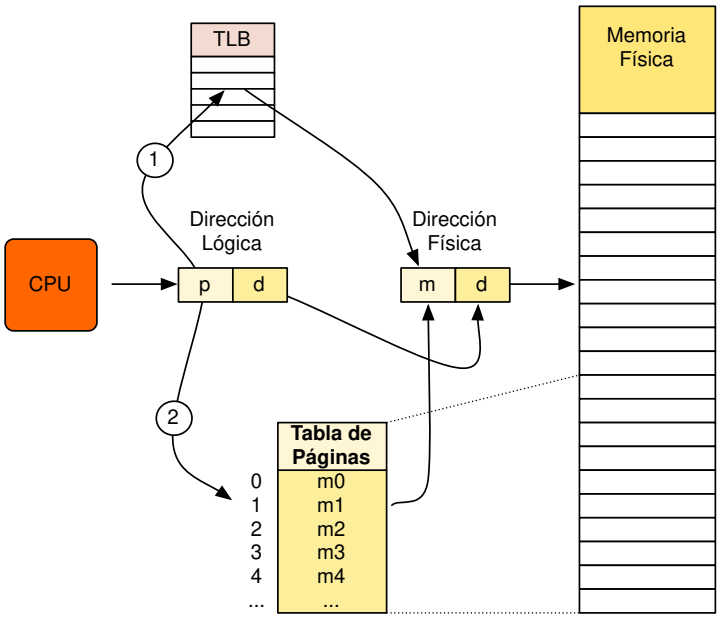

Figura 5.12: Esquema de paginación con TLB, en el paso 1 se consulta la TLB y si no se encuentra el número de página, en el paso 2 se consulta la tabla de páginas.

Si se accede a una página que no estaba en la TLB, se inserta. Si la TLB está llena, se debe desalojar una entrada usando algún algoritmo de reemplazo. Generalmente, en la TLB se pueden bloquear entradas para que no sean desalojadas (*pinning*).

La efectividad de la TLB depende de la tasa de aciertos (*hit ratio*), que es el porcentaje de veces que un página se encuentra en la TLB y no es necesario buscarla en la tabla de páginas. El *tiempo de acceso efectivo* a una dirección de memoria depende de la probabilidad de acierto (la probabilidad de que la traducción esté en la TLB) y el tiempo para llegar a la dirección física en cada caso (acierto o fallo):

$$T_{ef} = P_{acierto} * T_{acierto} + P_{fallo} * T_{fallo}$$

Por ejemplo, si la probabilidad de acierto es de 0.95, el tiempo de acceso en caso de estar en la TLB es 1 ciclo y el tiempo de acceso a la memoria es de 150 ciclos:

$$T_{ef} = P_{aciertoTLB} * (T_{accesoTLB} + T_{accesoMem.}) + P_{falloTLB} * (T_{accesoTLB} + T_{accesoPT} + T_{accesoMem.}),$$
$$= 0.95 * (1 + 150) + 0.05 * (1 + 150 + 150) = 159\, ciclos.$$

En este caso, la penalización es del 6% en el acceso (159 vs. 150 ciclos). En el caso de no tener TLB, la penalización siempre es del 100% porque se necesitan dos accesos (300 vs. 150 ciclos).

Hay que tener en cuenta que se debería limpiar la TLB (*TLB flush*) cuando hay un cambio de contexto. Este es otro de los factores que hacen que los cambios de contexto sean costosos. En el hardware moderno se intenta no borrar todas las entradas del la TLB usando distintas estrategias: metiendo información sobre el proceso al que pertenecen las entradas para evitar traducciones erróneas, evitando que entradas muy usadas sean eliminadas o permitiendo una eliminación selectiva de entradas.

Todo lo anterior, por supuesto es una simplificación, porque el acceso a la memoria no es directo, hay una jerarquía de cachés (L1, L2, L3, L4...), algunas direccionadas mediante direcciones físicas (detrás de la MMU) y otras mediante direcciones virtuales (antes de la MMU). Esto acelera los accesos a memoria en la tabla de páginas, que seguramente tenga sus entradas en la caché. Todos estos detalles hay que tenerlos en cuenta para estimar las velocidades de acceso a memoria (localidad en direcciones físicas vs. localidad en direcciones virtuales) y a la hora de hacer *flush* de las cachés en un cambio de contexto (no hará falta para las de direcciones físicas, sí para las virtuales, que cambian su significado).

Tabla de páginas invertida

El espacio de direcciones es muy grande (48 bits en AMD64) y cada proceso tiene el suyo. Esto conlleva un problema: las tablas de páginas son muy grandes y es difícil tener todas sus entradas contiguas en memoria.

Algunas arquitecturas (p. ej. PowerPC) usan **tablas de páginas invertidas** (IPT, ***I**nverted **P**age **T**able*), en las que tenemos al menos una entrada por cada marco de página (esto es, la tabla cubre el espacio de memoria física). En este caso, la tabla es la misma para todos los procesos (sólo hay una tabla). Una dirección lógica está formada por:

- PID del proceso.
- p: número de página.
- d: desplazamiento.

Cuando se necesita traducir, se busca en la tabla el par (PID, p). El problema ahora es que la tabla no está ordenada y la búsqueda es lenta. La solución es usar una tabla hash $(Hash(PID, p))$ con desbordamiento, como se muestra en la Figura 5.14. Esta tabla hash se implementa mediante hardware en estas arquitecturas. Otra forma más sencilla de entender las tablas de páginas invertidas es que son una únicamente una TLB con

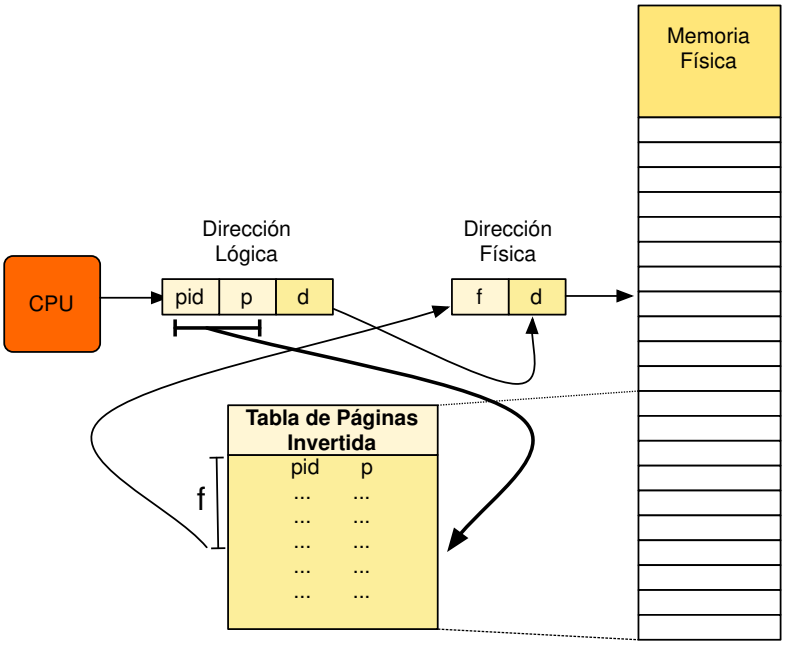

Figura 5.13: Esquema de paginación con IPT

respaldo en memoria, es decir, en lugar de tener tabla de páginas, lo que hay en memoria es la tabla de desbordamiento de la TLB. Este sistema es más sencillo y en sistemas con poca memoria y una TLB grande, disminuye mucho el número de accesos a memoria (la tabla hash esta poco llena y hay pocos desbordamientos que se traducen en fallos de la caché). Sin embargo, en los sistemas modernos con mucha memoria, va a haber gran cantidad de desbordamientos. A esto se suma que los accesos a memoria en la tabla hash van a tener poca localidad (es lo que hace una hash, aleatorizar los accesos), con lo que no hacen buen uso de los siguientes niveles de jerarquía (L1, L2...). Por estas razones, sólo se usan en sistemas pequeños embebidos (p. ej. el procesador PowerPC, que se usa todavía en algunos coches para el sistema de navegación).

Tabla de páginas multinivel

Para solucionar el problema del gran tamaño de la tabla de páginas, otras arquitecturas parten la tabla de páginas en niveles.

Veamos cómo funciona una de dos niveles. La dirección lógica está formada por $(p1, p2, d)$:

- p1: índice en la tabla exterior.
- p2: índice en la tabla interior.
- d: desplazamiento en la página.

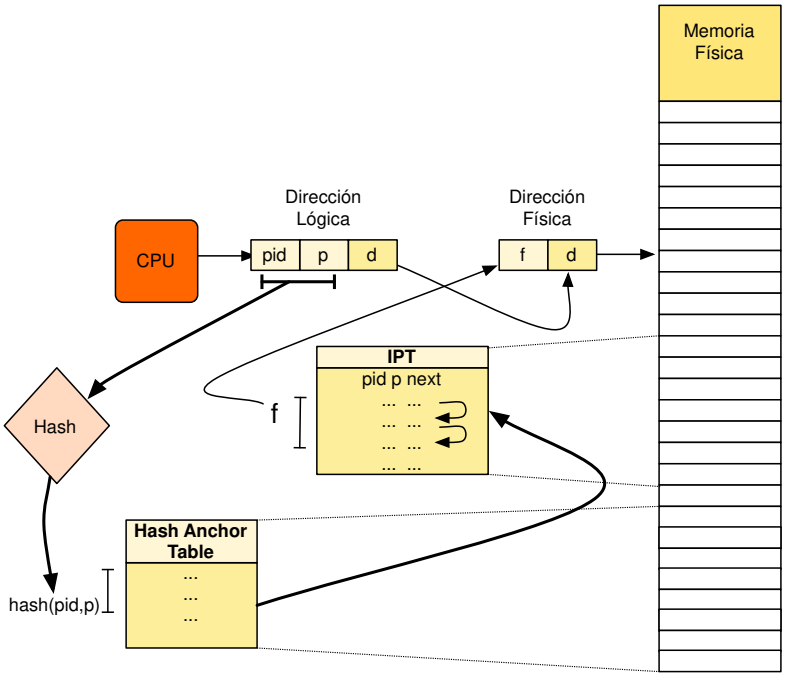

Figura 5.14: Esquema de paginación con IPT y hash

Para traducir una dirección, hay que buscar $p1$ en la tabla de primer nivel. Una vez encontrado el índice exterior, ya se tiene la tabla de segundo nivel. En ella se busca $p2$ para obtener la dirección base del marco de página. Luego se aplica el desplazamiento d en ese marco para acceder a la dirección física adecuada.

Por tanto, con una tabla de dos niveles necesitamos dos accesos a memoria adicionales por cada acceso real (tres en total para acceder a la memoria). Por ejemplo, supongamos de nuevo que la probabilidad de acierto es de 0.95, el tiempo de acceso en caso de estar en la TLB es 1 ciclo y el tiempo de acceso a la memoria es de 150 ciclos. Con una tabla de dos niveles el tiempo de acceso efectivo es:

$$\begin{aligned} T_{ef} &= P_{aciertoTLB} * (T_{accesoTLB} + T_{accesoMem.}) \\ &+ P_{falloTLB} * (T_{accesoTLB} + T_{accesoPT} + T_{accesoPT} + T_{accesoMem.}), \\ &= 0.95 * (1 + 150) + 0.05 * (1 + 150 + 150 + 150) = 166 \; ciclos \end{aligned}$$

En este caso, la penalización es del del 11% (166 vs. 150 ciclos). En el caso de no tener TLB, la penalización siempre es del 300% porque se necesitan tres accesos a la memoria (450 vs. 150 ciclos).

Como las tablas son muy grandes, normalmente tenemos más de dos niveles (provocando $N + 1$ accesos para un acceso real si tenemos que buscar en la tabla). Al final, el rendimiento depende del *hit ratio* de la TLB. Por ejemplo, los procesadores Intel i7

tienen la tabla de páginas de 4 niveles[11]. Parte de lo bueno de estas tablas de páginas es que los niveles superiores tienen una tasa de acierto en la caché L1 altísima, ya que se acceden contínuamente, con gran localidad temporal.

Otra cosa interesante es que las entradas de la tabla de página de niveles superiores pueden realizar varias funciones. Pueden apuntar a las tablas de nivel inferior o hacer de PTEs directamente apuntando a páginas más grandes (con bits que indican esto en donde iría el desplazamiento). Esto permite tener páginas de varios tamaños (para los diferentes niveles de la tabla de página), haciendo el esquema muy flexible.

5.3.2. Compartiendo páginas

Como ya sabemos, algunas partes del contenido de la memoria de varios procesos distintos pueden ser exactamente iguales. Por ejemplo, en dos procesos que están ejecutando el mismo programa, las páginas de sus segmentos de texto (instrucciones) serán exactamente iguales y vienen del mismo sitio: del fichero ejecutable. Además, esas páginas no se van a modificar durante la ejecución del programa. Son páginas de sólo lectura.

En este caso, no es necesario tener duplicadas esas páginas en memoria física. Los dos procesos pueden compartir los marcos de página que contienen ese segmento. Esto significa que las direcciones virtuales de esas páginas, en los dos procesos, se traducirán a las mismas direcciones físicas.

También se pueden compartir las partes que no son de sólo lectura mientras que sean iguales para ambos procesos. Una vez que uno de ellos tiene que modificar una de esas páginas, se duplica (se reserva un nuevo marco, se copia el contenido y se instala su traducción) y ya cada uno sigue con su copia (porque ya no son iguales). A esta estrategia se la denomina ***C**opy-**o**n-**W**rite* o COW.

Por ejemplo, cuando un proceso hace una llamada al sistema `fork`, se crea un proceso hijo que es idéntico al padre, como se puede ver en la figura 5.15. En ese momento, pueden compartir las páginas de sus segmentos. A medida que el padre o el hijo van modificando sus páginas, se hacen los duplicados y cada uno continua con su copia. En la figura 5.16 se puede ver un ejemplo en el que el padre modifica una página de memoria del segmento de datos justo después de `fork`.

Ten en cuenta que, en muchas ocasiones, no se va modificar una página. El caso más extremo (pero bastante habitual) es este: si el hijo va a llamar a `execv` después del `fork`, no habría servido para nada duplicar todas sus páginas en la creación del proceso, ya que ahora va a necesitar cambiar todo para ejecutar otro programa. Esta técnica es otro ejemplo de estrategia *perezosa*: hasta que no necesitamos de verdad hacer algo, no lo hacemos.

Por último, el sistema nos permite crear procesos que comparten memoria (*shared memory*). Eso significa que los procesos (o *hilos*) comparten algunos segmentos como el de texto, el de datos o el BSS. Esto es fácil de implementar: las páginas compartidas tienen la misma traducción en los procesos que las comparten.

[11] El registro `CR3` apunta a la tabla de primer nivel.

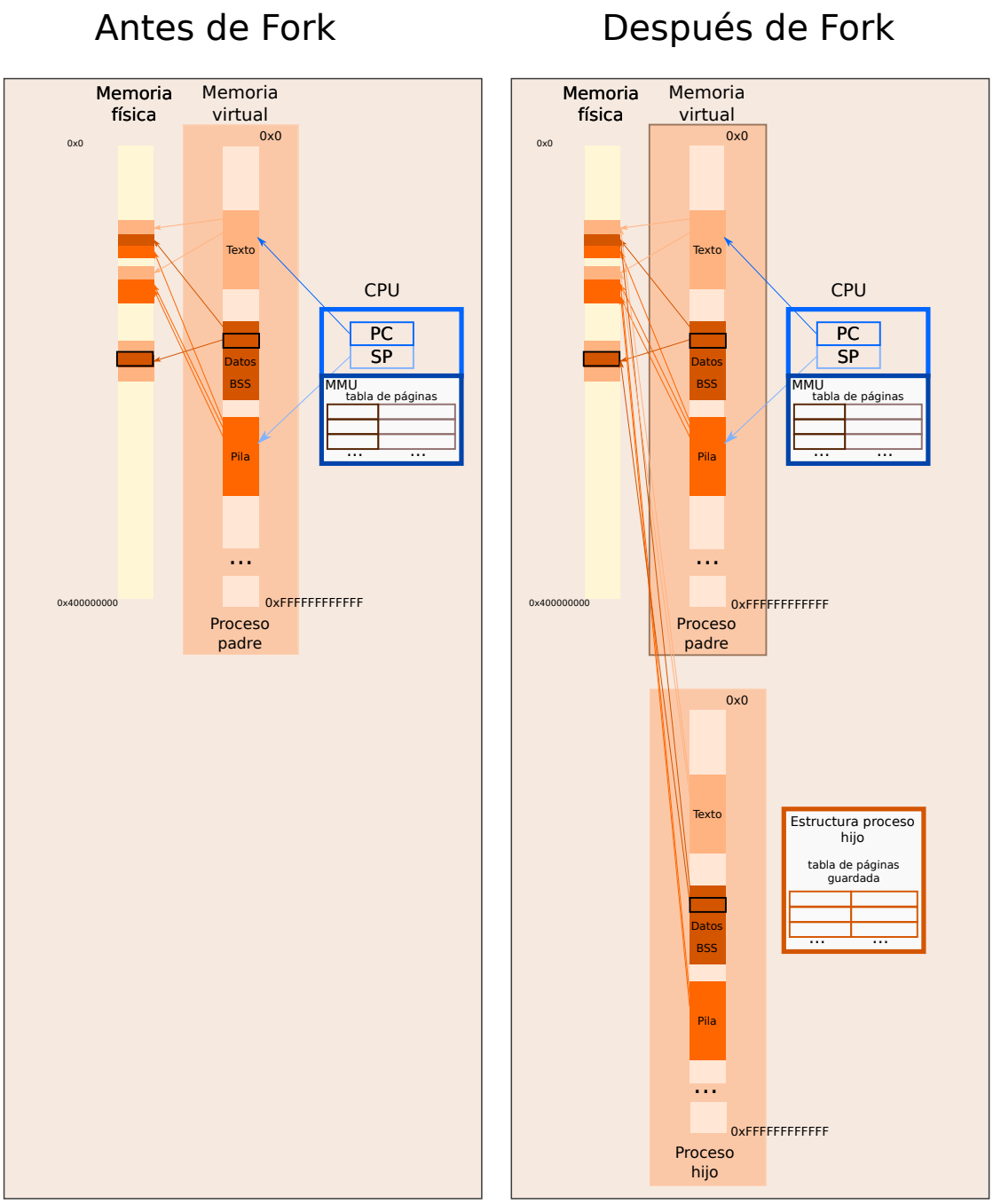

Figura 5.15: Justo después de fork, los procesos comparten todos los marcos de página.

5.3.3. Paginación en demanda

Al crear un proceso para ejecutar un programa, podríamos reservar marcos para toda su memoria, cargar todo del fichero ejecutable, inicializar las partes que no vienen de él

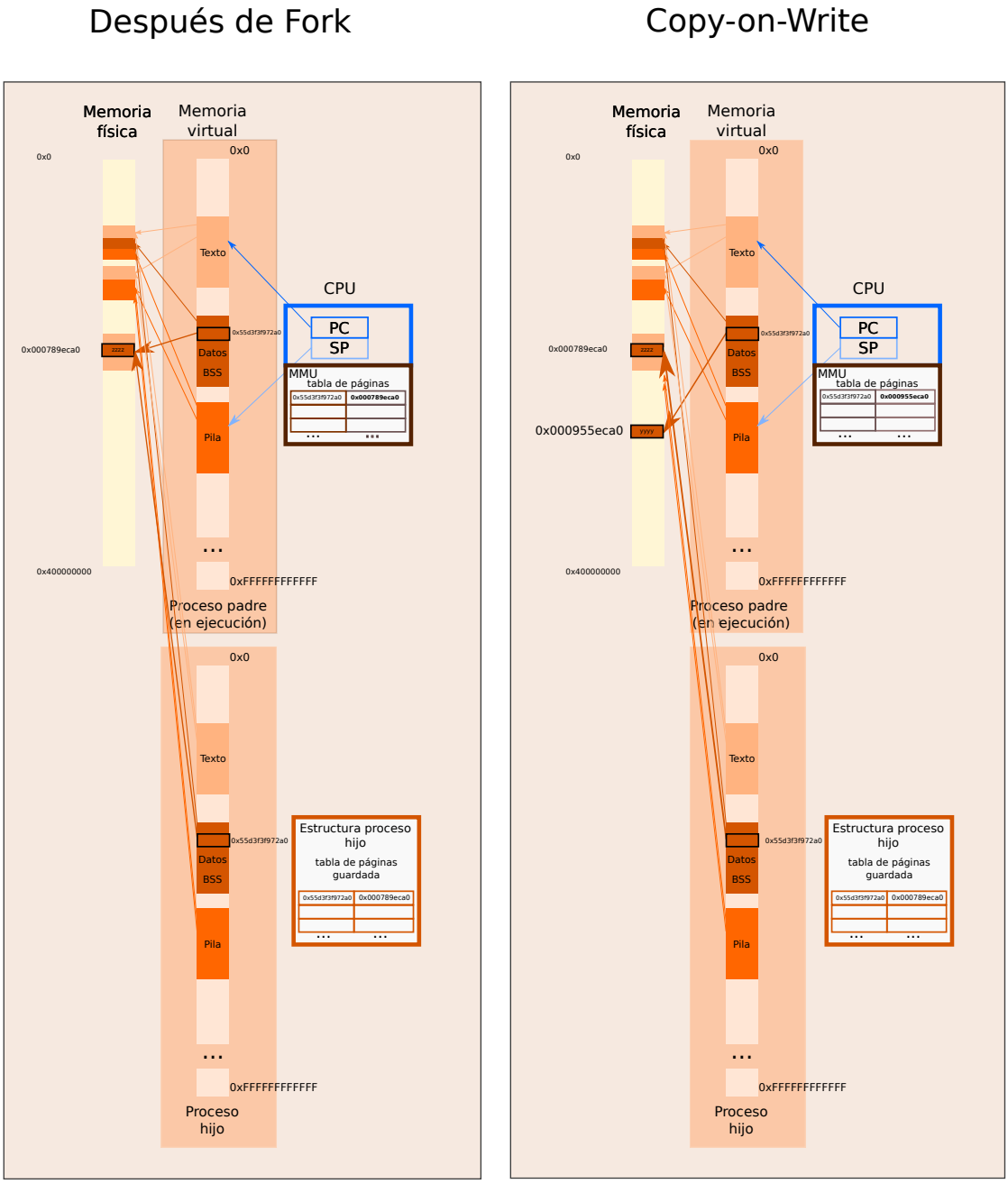

Figura 5.16: Cuando una página se modifica, su marco se duplica y se instala una nueva traducción.

(p. ej. BSS y pila) y ya cuando tuviésemos todo listo, ponerlo a ejecutar desde su punto de entrada.

Imagina ahora que es una de esas ejecuciones de tu programa en las que no has pasado

los argumentos necesarios y vas a salir en una de las primeras líneas del `main`, dando un fallo de uso del programa (lo que llamamos *usage*). ¿Para qué se ha gastado tiempo en reservar marcos de página, copiar su contenido e instalar todas las traducciones para todas sus páginas?

Los sistemas modernos aplican la paginación en demanda, que es otra estrategia perezosa. Consiste en buscar un marco de página e instalar la traducción para una página sólo cuando es estrictamente necesario, esto es, cuando tenemos un fallo de página porque la página no está todavía presente.

Esta estrategia consiste en ir trayendo las páginas del proceso a medida que se van teniendo fallos de página. Por tanto, la ejecución del proceso se va interrumpiendo a medida que hay fallos y se buscan marcos de página para él. Un inconveniente claro es que, si al final necesitamos buscar marcos para todas las páginas, habremos tardado más, pero no hemos hecho el trabajo en vano. Ya no tenemos un sobrecoste fijo en latencia al principio de la ejecución, y hacemos que fork sea (mucho) más rápido. En general, tenemos una mayor eficiencia en el uso del ancho de banda del bus de memoria, ya que las páginas a las que no se accede, no se copian.

Cuando un proceso de usuario ejecuta una instrucción que intenta acceder a una dirección de memoria sin traducción, salta una *trap* por fallo de página y el control pasa al kernel. Entonces el kernel maneja el fallo y mira qué ha pasado:

1. Se mira si la dirección pertenece al espacio de direcciones del proceso (si es de una región válida).

2. Si no pertenece, se manda una señal `SIGSEGV` al proceso (*segmentation fault*). ¿Te suena? Esto suele pasar cuando nuestro programa intenta acceder a una dirección que no tiene sentido (no tiene traducción).

3. Si es una dirección que sí pertenece al espacio de direcciones, se comprueban los permisos de la página y el tipo de acceso que se está haciendo (lectura, escritura, ejecución).

4. Si son correctos, se busca un marco para esa página. Si no, puede ser un fallo de protección que nos haría volver al paso 2, o COW, que nos permite continuar.

5. Si es necesario, se copia la página del fichero binario ELF al marco de página (p. ej. si es una página de TEXT o DATA), se inicializa (p. ej. BSS) o se copia de la original (si era COW y es una escritura).

6. Se modifica la tabla de páginas para incluir su traducción y poner su bit de presente. Si era COW (era una escritura), se cambia la protección de la página original.

7. En algún momento se vuelve a área de usuario en el contexto de ese proceso, con el contador de programa puesto para que se ejecute de nuevo la instrucción que ha generado el fallo de página. Ahora el proceso ya puede acceder a esa dirección de memoria virtual y ya no se provoca el fallo.

Esta es la razón por la que no debemos sobrescribir un ejecutable mientras que está en ejecución. Si lo hacemos, los procesos que estén ejecutando ese fichero no podrán conseguir las páginas cuando las necesiten.

5.3.4. Overcommitment

Al principio del tema hablamos del OOM (***O**ut **O**f **M**emory*). Cuando se llama a `malloc` y no tenemos memoria suficiente en el heap, se realiza una llamada al sistema `brk` para hacer crecer el segmento[12]. Algunos sistemas (paginación en demanda) lo pueden hacer crecer *sin haber reservado todavía* los marcos de página correspondientes (ni saber si quedan suficientes) y hacer que `malloc` devuelva una dirección de memoria virtual sin saber si tenemos memoria física para ella.

El marco correspondiente no se compromete hasta que se intenta usar la página correspondiente, esto es, hasta que se genera un fallo de página por ella. Esto se llama *overcommitment*. Lo puedes ver como el *overbooking* cuando compras un billete de avión: la compañía cuenta con que no todos los pasajeros van a acabar haciendo uso de sus billetes. Aquí pasa con los marcos de página. Con overcommitment, la llamada a `malloc` **no** va a fallar nunca (nunca retornará `NULL`) por falta de memoria física.

Lo mismo pasa con las variables globales sin inicializar (BSS) o cuando queremos crear una nueva región de memoria con la llamada `mmap`. Cuando se accede por primera vez a una dirección de memoria de una de esas páginas, se compromete su marco de página. Hasta entonces, no (esta es la base de la paginación en demanda). Hay estrategias intermedias, como contar cuantos quedan sin asignar. En *overcommitment* vamos "a lo loco", no se mira ni eso.

Entonces, ¿qué pasa cuando no quedan marcos de página, esto es, el sistema establece que se ha entrado en OOM? En ese caso, el sistema va a necesitar matar procesos para conseguir marcos de página. El *OOM-killer* se encarga de eso. ¿A quién mata? Los procesos tienen un *OOM score* que determina cuántas papeletas tienen para que les mate el OMM-killer cuando hagan falta marcos de página (cuanto más alto el *score*, más papeletas). En Linux, puedes verlo en este fichero:

```
/proc/$PID/oom_score
```

En Linux, el *overcommitment* es configurable a través de este otro fichero:

```
/proc/sys/vm/overcommit_memory
```

[12] Esto es, la región de memoria virtual del proceso que tiene anotada el kernel en sus estructuras de datos.

Puede funcionar en tres modos distintos: no permitirlo, permitirlo o permitirlo con una heurística que decide cuándo se puede y cuándo no (es el modo por omisión). Tienes toda la información en la página de manual *proc(5)*.

5.3.5. Ficheros y memoria

Como ya se comentó en el capítulo de ficheros, el sistema intenta usar toda la memoria RAM que puede para tener partes de los ficheros que está usando y así no pagar el coste de la entrada/salida del almacenamiento secundario. La *page cache* usa marcos de página que no necesitan los procesos para guardar datos de ficheros. Este tipo de cachés optimizan en gran medida el uso de ficheros. En un estudio realizado para el sistema 4.4BSD, se determinó que se ahorraban el 85% de las operaciones de entrada/salida.

Si estamos usando una caché, cuando se sincroniza el sistema (p. ej. con el comando `sync` o la llamada al sistema con el mismo nombre), se escriben los datos de los ficheros que están por escribir en el disco (páginas sucias, *dirty*), sincronizando la caché con su respaldo (i.e. los bloques del disco). Cuando se lee o se escribe un fichero, se están leyendo o escribiendo estas páginas de memoria usadas para la caché (a no ser que el fichero se haya abierto con la opción `O_DIRECT`, como vimos en su momento). Todo esto es transparente para el usuario.

La llamada al sistema `mmap` sirve para crear nuevas regiones de memoria en el espacio del proceso. Esto es, hacer que un rango de direcciones tenga ahora traducción. Se llama *mmap anónimo* cuando simplemente queremos una nueva región de memoria inicializada a cero.

Esta llamada al sistema también permite proyectar un fichero en memoria[13]. Esto consiste en crear una región nueva en la que encontraremos los contenidos de un fichero. Dicho de otra forma, podremos tratar con ese fichero como si fuese un array de bytes en memoria, sin tener que usar las llamadas al sistema de ficheros `read` y `write`. La memoria actúa como una caché del fichero. La llamada al sistema *msync(2)* efectua un flush de esta caché. Aunque puede ser cómodo para algunos programas, no sustituye a esas llamadas al sistema porque no es apto para tratar con ficheros sintéticos, ficheros que crecen, ficheros pequeños, sistemas de ficheros en red, etc.

Por ejemplo, el código de que se muestra a continuación proyecta el fichero que se le pasa como argumento en memoria, lo imprime por la salida y pasa todas sus letras a mayúsculas:

[13] Proyección nombrada, frente a la proyección anónima en el que la memoria no está respaldada por ningún fichero

Programa 5.3: mmap.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <pthread.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <sys/mman.h>
9  #include <err.h>
10 #include <ctype.h>
11
12 int
13 main(int argc, char *argv[])
14 {
15     int fd;
16     void *fmem;
17     char *array;
18     struct stat st;
19     int i;
20
21     if (argc != 2)
22         errx(EXIT_FAILURE, "usage: %s file", argv[0]);
23     fd = open(argv[1], O_RDWR);
24     if (fd < 0)
25         err(EXIT_FAILURE, "can't open file");
26     if (fstat(fd, &st) < 0)
27         err(EXIT_FAILURE, "can't stat file");
28     fmem =
29         mmap(NULL, st.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
30
31     if (fmem == MAP_FAILED) {
32         err(EXIT_FAILURE, "mmap failed");
33     }
34     array = (char *)fmem;
35     for (i = 0; i < st.st_size; i++) {
36         printf("%c", array[i]);
37         array[i] = toupper(array[i]);
38     }
39     if (munmap(fmem, st.st_size) < 0)
40         err(EXIT_FAILURE, "can't unmmap");
41     close(fd);
42     exit(EXIT_SUCCESS);
43 }
```

La proyección de ficheros en memoria se utiliza, por ejemplo, para cargar las secciones de los ficheros binarios ejecutables y las bibliotecas compartidas.

5.3.6. Intercambio

El intercambio (también llamado *swapping* o *paging*) consiste en mover partes (páginas) de memoria RAM a disco. En el caso de Linux, se usa una partición de tipo *swap* para esto (también se puede usar uno o varios ficheros, en lugar de una partición). Cuando el sistema requiere marcos de página, puede mover ciertas páginas a esa partición. Cuando el sistema necesita acceder al contenido de esas páginas, las tendrá que recu-

perar y asignarles marcos de páginas (y para ello, posiblemente enviar otras páginas al disco)[14].

Esto es costoso: tenemos que pagar el coste de la entrada/salida, tanto para salvarla a disco como para recuperarla. Ten en cuenta en el momento de escribir esto, un dispositivo Serial ATA (SATA-150) ofrece 1200 Mbit/s, mientras que la memoria RAM (DDR3-SDRAM en este caso) ofrece 136.4 Gbit/s.

Los procesos con privilegios pueden usar la llamada al sistema *mlock(2)* que permite que las páginas correspondientes a un rango dado nunca vayan a swap. La llamada *mlockall(2)* lo hace con toda la memoria del proceso.

Cuando se hace intercambio, se debe aplicar alguna política de reemplazo. Si es necesario traer una página de swap a memoria principal, hay que reservar un marco de página para ella y, si no hay ningún marco libre, tendremos que usar el marco que ocupa otra página (que habrá que salvar en swap). Lo ideal sería reemplazar las páginas a las que no se va a acceder en el futuro. El problema es que no podemos ver el futuro.

Si suponemos que la página que tiene menos probabilidad de ser usada en el futuro es la que hace más tiempo que no se usa (esto es, suponemos que existe una *localidad temporal* en el acceso), lo óptimo sería reemplazar esa página. Esa política se llama **LRU** (***L**east **R**ecently **U**sed*). Una posible implementación sería incrementar en la tabla de páginas el valor de un contador global cuando se accede a la página y cuando se necesite, se reemplaza la página con el contador más bajo. El problema es que esto necesita apoyo del hardware para actualizar el contador, porque no es admisible una interrupción por cada acceso a memoria para que el sistema operativo se encargue de apuntar el valor del contador en la tabla de páginas. Malas noticias: el hardware disponible no puede hacer eso.

El algoritmo **FIFO** es muy simple: se reemplaza la página más vieja, la que lleva más tiempo en un marco. El problema es que eso no tiene nada que ver con cuánto se usa la página o si ha tenido accesos recientes (y, por tanto, pensamos que se puede usar en un futuro cercano). Además, con esta política podemos sufrir lo que se conoce como la anomalía de Belady: siendo $N > M$, el número de fallos para N marcos puede ser mayor que para M marcos (esta anomalía no es exclusiva de la política FIFO, también se da en otras). En otras palabras, ¡comprar más RAM empeora el rendimiento!

Lo que podemos hacer es utilizar políticas que se aproximen a LRU. Por ejemplo, la política **NRU** (***N**ot **R**ecently **U**sed*) selecciona una página que no tiene accesos recientes. Es algo binario: ¿desde un momento dado se ha accedido la página? Para esto, basta con que el hardware te proporcione un único bit por página que se pone a 1 cuando se usa. Ese bit se denomina *bit de referencia*. Periódicamente, se ponen todos bits de referencia a 0. Después, a la hora de reemplazar, se mira qué páginas lo tienen a 0 todavía y se escoge una de ese grupo aleatoriamente.

Adicionalmente, se puede mantener otro bit que indique si la página está sucia o no lo está, el *bit de sucio* (*dirty*). Una página está sucia si se ha modificado su contenido. Si está sucia, hay que descargar su contenido a almacenamiento secundario si la desaloja-

[14] Esto se usa, por ejemplo en la hibernación de un portátil. Se salvan todas las páginas a disco y se apaga el dispositivo.

mos. En otro caso, nos lo podremos ahorrar (porque no ha cambiado desde que se copió su contenido al marco de página). Siempre será mejor elegir una página limpia, porque así nos ahorraremos el tiempo de entrada/salida necesario para copiar sus datos en almacenamiento secundario. Si estamos usando los dos bits (usado/no usado, limpia/sucia), siempre será mejor una sin accesos recientes. Si no hay ninguna sin usar recientemente, siempre es preferible una limpia que una sucia. El principal problema de NRU es que no ordena las páginas, sólo las mete en dos conjuntos.

Si tenemos disponible el bit de referencia, también podemos usarlo para tener un algoritmo **FIFO con segunda oportunidad**. Ahora no se pone el bit de referencia a 0 periódicamente. Cuando el orden FIFO indica que le toca a una página, si tiene el bit de referencia a 1, entonces se vuelve a colocar al final de la cola, pero con ese bit a 0. Esto es, esa página se salva. Ten en cuenta que, si todas las páginas se usan, el algoritmo degenera en FIFO puro. Esto mejora el algoritmo FIFO, ya que ahora las páginas viejas pero usadas se salvan de ser reemplazadas.

Esta misma idea se puede implementar, pero usando un array circular y utilizando un índice que lo va recorriendo. De esta forma, cuando necesitamos reemplazar una página, miramos a cuál se refiere el índice: si tiene el bit de referencia a 1, lo ponemos a 0 y pasamos al siguiente, así hasta encontrar uno a 0. Cuando se encuentra uno a 0, se reemplaza esa página y se avanza el índice. Este es el algoritmo del **reloj**. El índice es la manecilla del reloj.

Existe una variante de este último algoritmo, el **reloj con dos manecillas**. Ahora se usan dos índices para recorrer el array circular, esto es, dos manecillas. Una va siempre delante, entre ellas hay una distancia constante y van avanzando al mismo ritmo. La primera manecilla va poniendo a 0 los bits de referencia que se encuentra a 1. La segunda manecilla elige víctima, que será la página que se encuentre con el bit de referencia a 0. Por tanto, si en el intervalo entre el paso de ambas manecillas no se ha accedido a una página, entonces esa página será la víctima.

Existen otros algoritmos de reemplazo y mucha información sobre este asunto en la bibliografía [1, 2, 3].

La asignación de marcos puedes ser **local** o **global**. Local significa que se desaloja una página del mismo proceso que ha tenido el fallo de página. Los procesos tienen un número de marcos dados y con esos se las tienen que apañar. Dos ejecuciones iguales tendrán el mismo número de fallos de página. Global significa que se puede desalojar una página de otro proceso (se *roba* el marco de página). En este caso, se puede robar a procesos menos usados o con menos prioridad que el proceso que ha tenido el fallo.

La asignación puede ser **equitativa**, si a todos los procesos se les asigna el mismo número de marcos, o **proporcional** a la memoria que necesita el proceso para ejecutar (no todos los procesos necesitan la misma cantidad de memoria). También se puede premiar por la prioridad del proceso, etc.

El paginador (parte del kernel que se dedica a esto) puede aplicar otras técnicas para optimizar la paginación:

- Mantener un conjunto de marcos libres de reserva para cuando hagan falta, no esperar a que se llene toda la memoria.

- Desalojar marcos de páginas de la caché de páginas que ya no hagan falta, por ejemplo, los dedicados a los ficheros binarios que no está ejecutando ningún proceso actualmente.
- En ratos ociosos, se pueden escribir las páginas sucias a disco y marcarlas como limpias.
- Se pueden desalojar páginas dejando el contenido en el marco (aunque esté libre, se queda con el contenido). Si después hay que traer la misma página, no hace falta realizar ninguna operación de entrada/salida, ya tenemos el contenido allí.

Cuando un sistema con paginación (*paging*) está saturado, puede entrar en *thrashing*. Esto sucede cuando el sistema gasta más recursos en la propia paginación que en procesamiento útil para servir al usuario. Por ejemplo, puede suceder cuando hay asignación global y muchos procesos concurrentes. El efecto es que los procesos se roban marcos entre ellos continuamente. Cada vez que hay cambios de contexto, hay muchos fallos de página y se necesita intercambiar continuamente. Cuando el sistema entra en *thrashing*, veremos que los procesos no progresan y hay mucha carga de entrada/salida (páginas que van y vienen del área de intercambio).

La solución a este problema es tener marcos suficientes para el *conjunto de trabajo* de los procesos. El conjunto de trabajo de un proceso es el conjunto de páginas en las últimas Δ referencias, siendo Δ la *ventana del conjunto de trabajo* [2]. Si una página está en uso activo, entonces está en el conjunto de trabajo. Si no lo está, será eliminada del conjunto de trabajo en Δ unidades de tiempo desde su última referencia. Dicho de una forma simple, *el conjunto de trabajo* es el conjunto de páginas *que están ahora en uso* en el proceso. Una vez más, nos basamos en la localidad temporal.

El sistema intenta tener asignado a un proceso un número de marcos de página igual a su conjunto de trabajo. Cuando se crea un proceso nuevo, el sistema puede estimar si se entrará en *thrashing* si tiene en cuenta el conjunto de trabajo de todos los procesos.

5.3.7. /proc/meminfo

En Linux, el fichero `/proc/meminfo` nos proporciona mucha información sobre la gestión de memoria. Este fichero sintético ofrece:

- `MemTotal`: memoria física usable (total menos el binario del kernel).
- `MemFree`: memoria libre.
- `MemAvailable`: estimación de memoria disponible para nuevos procesos, sin contar swap.
- `Buffers`: bloques de dispositivos cacheados.
- `Cached`: tamaño de la *page cache* (no incluye lo que hay en swap).
- `Dirty`: memoria sucia que tiene que bajar a disco.

- ...

En la página de manual *proc(5)* tienes toda la información.

5.4. Ejercicios no resueltos

1. Escribe un programa que tenga un leak de memoria. Ejecútalo bajo `valgrind` y observa su salida.

2. Escribe un programa que atraviese un puntero erróneo. Por ejemplo:

   Programa 5.4: crash.c

   ```
   1  #include <stdio.h>
   2  #include <stdlib.h>
   3
   4  int
   5  main(int argc, char *argv[])
   6  {
   7      char *p;
   8
   9      p = (char *)0x18;
   10     printf("s.x :%d\n", *p);
   11 }
   ```

   Depúralo con `gdb`. Mueve el error a una función. Depúralo con `gdb` de nuevo, imprime la pila. Consigue que genere un *core* (mira el comando `ulimit` y la página de manual de `Apport`: `apport-cli(1)`). †∗

3. Instala el paquete `wamerican`. Este paquete incluye un fichero con palabras:

   `/usr/share/dict/words`

   Escribe un programa en C construya una tabla hash con las palabras del fichero y lea palabras de la entrada estándar y diga si están en el fichero o no. Ejecútalo bajo `valgrind` y observa su salida. ††∗∗

4. Escribe un programa que haga `fork` y el hijo llame a la función de biblioteca `sleep` por un tiempo largo e imprima el pid de ambos procesos. Inspecciona los ficheros `/proc/XXX/maps` de ambos procesos. †∗

5. Programa un *bump allocator* que reparta un array estático muy grande de memoria (las funciones se llamarán `mymalloc` y `myfree` de momento para no colisionar con las del sistema). †∗

6. Programa otra versión del allocator que utilice una lista doblemente enlazada con headers y footers y tamaño mínimo. Recuerda que las estructuras de datos van dentro del array (que representa la memoria). † † †∗ ∗ ∗

7. Modifica el anterior allocator que utilice *Quick Fit* para los fragmentos de 64, 128 y 256 bytes. † † †∗ ∗ ∗

8. Escribe un programa reserve y libere memoria a menudo (o coge uno que ya escribiste) y mide las diferencias de rendimiento entre los diferentes asignadores utilizando `gprof`. † † ∗

9. Renombra tus funciones como `malloc` y `free`. Compila tus allocators como una biblioteca dinámica:

```
$ gcc -shared -fPIC -o myalloc.so myalloc.c
$
```

   Ejecuta diferentes comandos usando tus allocators y prueba el rendimiento de los mismos con tus diferentes allocators. Mira cómo de rápido es cada uno, cuanto tarda en quedarse sin memoria, etc. ††∗∗

```
$  LD_PRELOAD=./myalloc.so ls
$  LD_PRELOAD=./myalloc.so top
$
```

10. Averigua utilizando lo siguiente de tu sistema:
    - Tamaño total de la RAM (`/proc/meminfo`)
    - De donde viene la swap (`swapon -s`, `/proc/swaps` o `free -m`)
    - Qué tamaño de página utiliza el sistema `getconf PAGE_SIZE`.

    †∗

11. Busca el manual para programadores del sistema operativo de la CPU de tu máquina (por ejemplo [27]). Intenta averiguar cómo funciona su tabla de páginas. Haz lo mismo con tu *smartphone*. † † †∗∗

6 Comunicación entre procesos

Los procesos que no comparten memoria necesitan mecanismos para comunicarse entre ellos. Este capítulo se centrará en varios de los mecanismos de comunicación entre procesos (IPC, ***I**nter-**P**rocess **C**ommunication*) más populares en los sistemas de tipo Unix: los *pipes*, los *sockets* y las señales.

6.1. Pipes

En el capítulo de shell scripting ya aprendimos a usar los *pipes* (tuberías) de forma efectiva en el intérprete de comandos. Ya conocemos el estilo Unix: tener pequeños programas (filtros) que leen datos por su entrada estándar, los procesan y los escriben por su salida estándar. Los *pipes* nos permiten concatenar programas conectando la entrada de uno a la salida de otro. Ahora nos centraremos en su uso realizando llamadas al sistema desde un programa. Así podremos entender, entre otras cosas, como está implementada la shell.

Un *pipe* es un buffer en área de kernel que sirve para comunicar procesos. En Unix, los *pipes* son *simplex*: tiene dos extremos, uno de escritura y otro de lectura. Todo lo que se escribe en el extremo de escritura se lee del extremo de lectura. Cada uno de esos extremos se comporta como un fichero abierto y múltiples procesos pueden estar leyendo y/o escribiendo de/en ellos. Algunos sistemas tienen *pipes full-duplex*[1], en los que se puede leer y escribir en los dos extremos simultáneamente. En Linux, los *pipes* son como en Unix (*simplex*).

La llamada al sistema `pipe` (ver *pipe(2)*) crea una nueva tubería. Esta llamada al sistema tiene un único parámetro, que es un array de enteros. En ese array dejará los descriptores de fichero correspondientes a los dos extremos del *pipe* creado. El extremo en la posición 0 es el de lectura. El de la posición 1 es el extremo de escritura. En caso de fallo, la llamada retorna -1.

Para que un proceso (padre) pueda compartir el *pipe* con otro proceso (hijo), siempre se creará el *pipe* antes de crear el proceso hijo. Recuerda que la tabla de descriptores del proceso hijo es una copia exacta a la del padre en el momento de su creación y ambos comparten las estructuras de los ficheros abiertos en el momentos de creación.

¿Qué pasa cuando ejecutamos estos comandos en la shell?

```
 $ ls *.txt | wc -l
12
 $ cat *.txt | grep 'pepe' |  wc -l
32
 $
```

Por cada *pipeline* que ejecutamos, la shell crea tantos procesos como comandos tenga el *pipeline*, pero antes creará tantos *pipes* como barras verticales encuentre (número de

[1] Por ejemplo Plan 9. En Linux, se podrían usar, si se necesita que sean *full-duplex*, sockets de comunicación de procesos tipo unix, *unix(7)*.

procesos menos uno)[2]. Después configurará la entrada y la salida de cada uno de los procesos para leer y escribir en el extremo indicado (ver figuras 6.1 y 6.2).

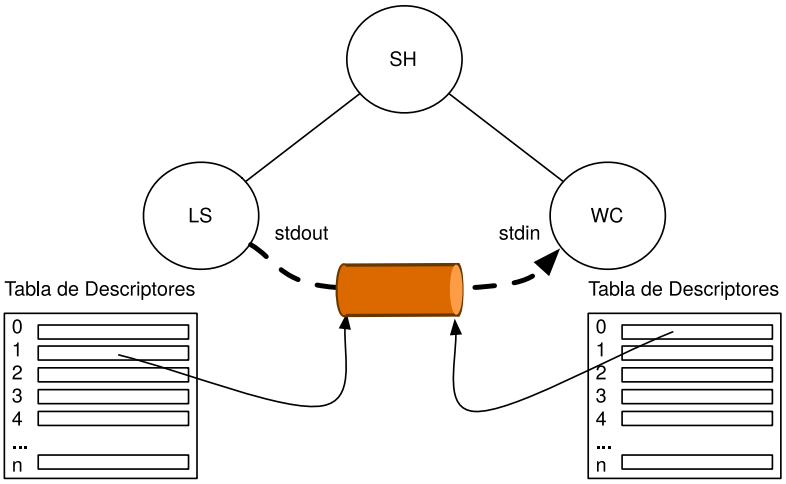

Figura 6.1: Pipeline con dos procesos.

Siempre debemos cerrar el extremo del *pipe* que no necesitemos usar. Si el proceso va a leer del *pipe*, debería cerrar el extremo de escritura, y viceversa. Es importante que tampoco se nos olvide cerrar un extremo después de duplicarlo (p. ej. si hemos hecho que el *pipe* sea la entrada o salida estándar del proceso). En general, cuando hacemos eso (p. ej. para ejecutar otro programa después), no queremos dejar abierto el descriptor duplicado. Como veremos un poco más adelante, dejarlo abierto nos va a causar problemas.

Veamos un ejemplo sencillo. En el siguiente programa se crea un proceso hijo. El proceso padre envía al proceso hijo una cadena de caracteres a través del *pipe*. Nótese que el hijo sólo lee una vez del *pipe*, con un buffer lo suficientemente grande para que entre la cadena. Ya sabemos que si queremos leer un número de bytes dado, tenemos que insistir llamando a `read` tantas veces como haga falta, porque puede haber lecturas cortas. En este ejemplo vamos a ignorar este problema (arriesgándonos a leer menos, es mejor no hacer esto en código real):

[2]Para no tener que cerrar descriptores de fichero innecesariamente en los hijos, no será exactamente en este orden, irá creando los pipes según los necesite.

Programa 6.1: pipe.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

enum {
    Bsize = 1024,
};

void
child(int *fd)
{
    char buf[Bsize];
    long n;

    close(fd[1]);
    n = read(fd[0], buf, Bsize);    //ojo, falta un bucle, mal escrito
    if (n < 0)
        err(EXIT_FAILURE, "cannot read from pipe");
    if (n > 0) {
        buf[n] = '\0';
        printf("Child: %s\n", buf);
    }
    close(fd[0]);
}

int
main(int argc, char *argv[])
{
    int fd[2];
    char *str = "I am your father";

    if (pipe(fd) < 0) {
        err(EXIT_FAILURE, "cannot make a pipe");
    }
    switch (fork()) {
    case -1:
        err(EXIT_FAILURE, "cannot fork");
    case 0:
        child(fd);
        exit(EXIT_SUCCESS);
    default:
        close(fd[0]);
        if (write(fd[1], str, strlen(str)) != strlen(str)) {
            err(EXIT_FAILURE, "error writting in pipe");
        }
        close(fd[1]);
    }
    exit(EXIT_SUCCESS);
}
```

Al ejecutar ese código, vemos como el proceso hijo escribe por su salida estándar la cadena que ha leído del *pipe*:

```
 $ ./pipe
Child: I am your father
 $
```

Veamos otro ejemplo sencillo en el que el hijo ejecuta otro programa. En este caso, ejecuta un `wc -w`. Por tanto, el hijo contará el número de palabras que lee de su entrada y acabará. El padre esperará a que el hijo acabe.

Programa 6.2: pipewc.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <err.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10 #include <sys/wait.h>
11
12 int
13 main(int argc, char *argv[])
14 {
15     int fd[2];
16     char *str = "I am your father\n";
17     int status;
18
19     if (pipe(fd) < 0) {
20         err(EXIT_FAILURE, "cannot make a pipe");
21     }
22     switch (fork()) {
23     case -1:
24         err(EXIT_FAILURE, "fork failed");
25     case 0:
26         close(fd[1]);
27         if (dup2(fd[0], 0) < 0) {
28             err(EXIT_FAILURE, "dup failed");
29         }
30         close(fd[0]);
31         execl("/usr/bin/wc", "wc", "-w", NULL);
32         err(EXIT_FAILURE, "exec failed");
33     default:
34         close(fd[0]);
35         if (write(fd[1], str, strlen(str)) != strlen(str)) {
36             err(EXIT_FAILURE, "error writting in pipe");
37         }
38         close(fd[1]);
39         if (wait(&status) < 0) {
40             status = EXIT_FAILURE;
41         }
42     }
43     exit(status);
44 }
```

Al ejecutar el programa veremos la salida del hijo, que en total ha contado cuatro

palabras (la línea que ha leído de su entrada estándar, que es el *pipe*):

```
 $ ./pipe
4
 $
```

En general, cuando se lee de un *pipe* pueden pasar varias cosas:

- Que haya datos disponibles para leer en el *pipe*: en este caso se leerán los datos disponibles inmediatamente.
- Que no haya datos disponibles para leer, pero haya procesos en el otro extremo que pueden escribir cosas en el *pipe*. En este caso, la lectura se queda bloqueada hasta que haya datos disponibles para leer.
- Que no haya datos disponibles en el *pipe* y no haya ningún proceso con el extremo de escritura abierto. En este caso, la lectura retornará 0 bytes (i.e. se ha llegado al final del fichero).

Por esto es importante cerrar todos los descriptores que no queramos usar: si nos dejamos un extremo de escritura abierto en algún proceso, los lectores nunca van a llegar a final de fichero (¡queda alguien al otro lado que todavía puede escribir!).

El *pipe* tiene un buffer limitado, se puede llenar. Cuando se escribe en un *pipe*:

- Si hay espacio suficiente en el buffer para los datos, se escriben.
- Si no hay espacio, la escritura se quedará bloqueada hasta que haya espacio para que se realice.
- Si se intenta escribir en un *pipe* que no tiene ningún proceso con el extremo de lectura abierto, el proceso que intente realizar dicha escritura recibirá una señal (que por omisión matará el proceso). Tiene sentido: ningún proceso podría sacar los datos del buffer.

Una escritura en un *pipe* no se va a *intercalar* con otra escritura en el *pipe* siempre que esté por debajo de un tamaño dado (constante `PIPE_BUF`), que suele ser de 4KB en Linux (el estándar POSIX establece un tamaño mínimo de 512 bytes)[3].

En los *pipes* de Unix no se respetan los *límites de escritura* (también se llaman *delimitadores de escritura*). Esto significa que si una escritura metió N bytes en el *pipe*, una lectura puede devolver más de esos N bytes[4] (si hubo una escritura posterior). Hay otros sistemas que sí garantizan los límites.

[3] Ojo, no confundas esto con los delimitadores o límites de escritura que veremos a continuación, que no se garantizan en ningún caso.

[4] En Linux otra llamada, `pipe2` (ver *pipe2(2)*) que permite pasar flags. Hay una flag, `O_DIRECT` que hace que el *pipe* preserve delimitadores salvo para un caso especial: las escrituras de tamaño cero. Las escrituras de tamaño cero se pierden en cualquier caso: las escrituras de tamaño cero provocan un `return` temprano en la función que maneja el write en el sistema de ficheros que implementa los pipes en el kernel.

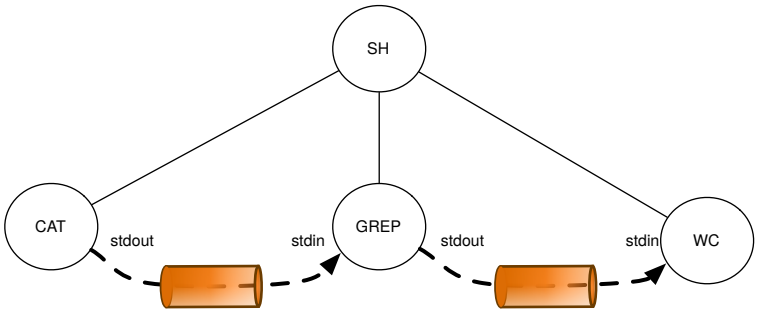

Figura 6.2: Pipeline con tres procesos.

Los *pipes* están pensados para sincronizar a los procesos lectores y escritores. Ejemplos de esto se ven en las figuras 6.3 6.4 y 6.5 Si los escritores van muy rápido, se quedarán bloqueados intentando escribir si se llena el *pipe*. Si los lectores son más rápidos, se quedarán bloqueados intentando leer del *pipe*. Es muy importante entender esto. Cuando tenemos un *pipe* para comunicar dos procesos, ambos deberían leer y escribir concurrentemente, no secuencialmente.

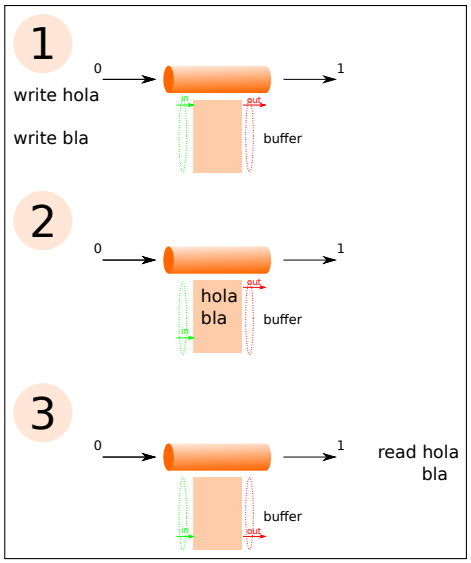

Figura 6.3: Ejemplo de escritura en un pipe, se guarda en el buffer y más adelante se lee.

Por ejemplo, nunca se debe crear un *pipe*, escribir en él y crear luego el proceso que leerá de él. En ese caso, ¿qué pasa si se llena el *pipe*? Habrá un **interbloqueo** (*deadlock*). El proceso que escribe se queda bloqueado intentando escribir en el *pipe*, pero está lleno. El proceso que va a leer todavía no está leyendo, porque no ha sido creado. Por tanto, no progresa ninguno de los dos. El siguiente programa ilustra el problema (recibe un único con el número de líneas que va a escribir en el *pipe*):

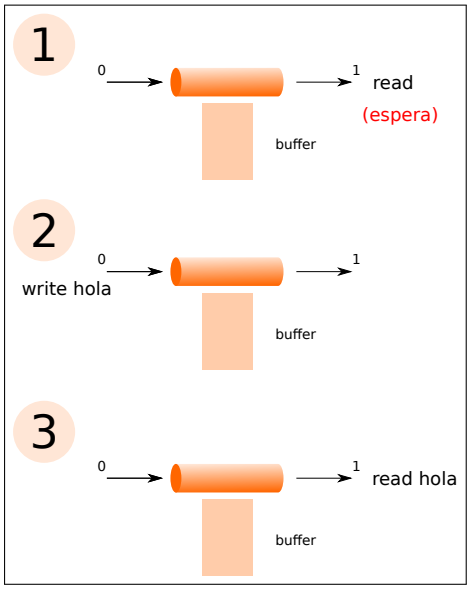

Figura 6.4: Ejemplo de lectura de un pipe, la lectura se bloquea esperando datos.

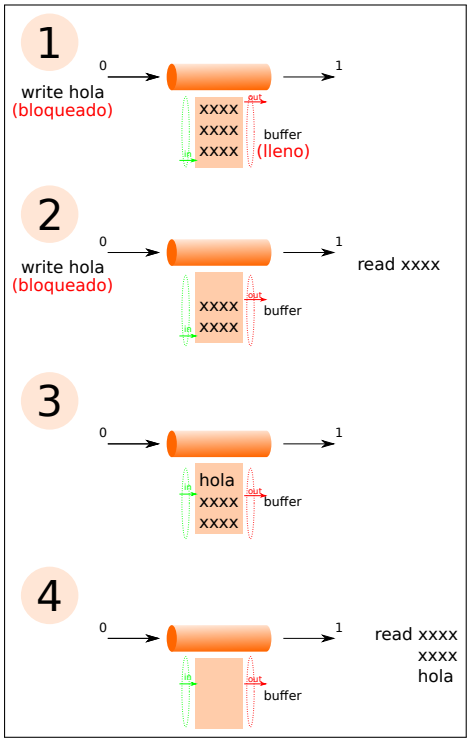

Figura 6.5: Ejemplo de escritura en un pipe con el buffer lleno. Se bloquea y más adelante una lectura lo desbloquea.

Programa 6.3: deadlock.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <err.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10 #include <sys/wait.h>
11
12 enum {
13     Linelen = 256,
14 };
15
16 int
17 main(int argc, char *argv[])
18 {
19     char line[Linelen];
20     int fd[2];
21     int i;
22     int status;
23     int lines;
24
25     if (argc != 2) {
26         errx(EXIT_FAILURE, "number of lines not found");
27     }
28     lines = atoi(argv[1]);
29     if (lines <= 0) {
30         errx(EXIT_FAILURE, "invalid number");
31     }
32     if (pipe(fd) < 0)
33         err(EXIT_FAILURE, "cannot make a pipe");
34     for (i = 0; i < lines; i++) {
35         snprintf(line, Linelen, "This is line %d\n", i);
36         if (write(fd[1], line, strlen(line)) != strlen(line)) {
37             err(EXIT_FAILURE, "write failed");
38         }
39     }
40     switch (fork()) {
41     case -1:
42         err(EXIT_FAILURE, "cannot fork");
43     case 0:
44         close(fd[1]);
45         if (dup2(fd[0], 0) < 0) {
46             err(EXIT_FAILURE, "dup failed");
47         }
48         close(fd[0]);
49         execl("/usr/bin/wc", "wc", "-l", NULL);
50         err(EXIT_FAILURE, "exec failed");
51     }
52     close(fd[0]);
53     close(fd[1]);
54     if (wait(&status) < 0) {
55         status = EXIT_FAILURE;
56     }
57     exit(WEXITSTATUS(status));
58 }
```

Cuando ejecutamos este programa con 100 líneas, funciona correctamente. También cuando lo ejecutamos con 1000 líneas. Pero, ¿y si intentamos que escriba 10000 líneas en el *pipe*?

```
 $ ./deadlock 100
100
 $ ./deadlock 1000
1000
 $ ./deadlock 10000
^Z
[1]+  Stopped                 ./deadlock 10000
 $ bg
[1]+ ./deadlock 10000 &
 $ ps aux | fgrep ./deadlock
esoriano  6976  0.0  0.0   4376   736 pts/3    S    21:25   0:00 ./deadlock 10000
 $
```

En ese caso el programa se queda bloqueado. Si nos fijamos en su estado, vemos que está esperando algún evento[5]. ¿Qué evento está esperando? Está esperando a que acabe la llamada a `write`. Se ha llenado el *pipe* y la escritura se quedará bloqueada hasta que haya hueco en el buffer[6] del *pipe* (como se ve en la figura 6.5). Pero hay un problema: el proceso hijo, que es el que lee del *pipe*, todavía no existe. Por tanto, nadie va a sacar datos del *pipe*. El proceso se quedará bloqueado para siempre. Como hemos dicho antes, **siempre debemos leer y escribir en el *pipe* concurrentemente**.

Una versión corregida de este programa sería:

[5]Estado S: *interruptible sleep* esperando a que se complete un evento.

[6]El buffer es circular: mantiene un puntero con lo que escribe y otro con lo que lee (*in* y *out*). Ambos punteros cuando llegan al final comienzan por el principio de nuevo.

Programa 6.4: nodeadlock.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <err.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10 #include <sys/wait.h>
11
12 enum {
13     Linelen = 256,
14 };
15
16 int
17 main(int argc, char *argv[])
18 {
19     char line[Linelen];
20     int fd[2];
21     int i;
22     int status;
23     int lines;
24
25     if (argc != 2) {
26         errx(EXIT_FAILURE, "number of lines not found");
27     }
28     lines = atoi(argv[1]);
29     if (lines <= 0) {
30         errx(EXIT_FAILURE, "invalid number");
31     }
32     if (pipe(fd) < 0)
33         err(EXIT_FAILURE, "cannot make a pipe");
34     switch (fork()) {
35     case -1:
36         err(EXIT_FAILURE, "cannot fork");
37     case 0:
38         close(fd[1]);
39         if (dup2(fd[0], 0) < 0) {
40             err(EXIT_FAILURE, "dup failed");
41         }
42         close(fd[0]);
43         execl("/usr/bin/wc", "wc", "-l", NULL);
44         err(EXIT_FAILURE, "exec failed");
45     }
46     close(fd[0]);
47     for (i = 0; i < lines; i++) {
48         snprintf(line, Linelen, "This is the line %d\n", i);
49         if (write(fd[1], line, strlen(line)) != strlen(line)) {
50             err(EXIT_FAILURE, "write failed");
51         }
52     }
53     close(fd[1]);
54     if (wait(&status) < 0) {
55         status = EXIT_FAILURE;
56     }
57     exit(WEXITSTATUS(status));
58 }
```

Ahora, si ordenamos escribir 10000 líneas, ya no se queda bloqueado:

```
 $ ./nodeadlock 10000
10000
 $
```

¿Qué pasaría si comentásemos la línea 38 del programa (la llamada a `close` para cerrar el extremo de escritura en el hijo)?

```
 $ ./nodeadlock 10
```

Otra vez tenemos un interbloqueo. ¿Por qué? El programa `wc` está leyendo de su entrada, sigue leyendo porque no ha llegado al final del fichero. ¿Hay alguien con el extremo de escritura abierto? Sí, él mismo.

Algunos programas usan un *pipe* para almacenar algún dato sensible, ya que es un buffer en área de kernel (p. ej. para almacenar una contraseña). Si se necesita hacer algo como eso, hay que ser muy cuidadoso para no terminar con un interbloqueo (si se llena el buffer). No es una práctica recomendable en general.

6.2. Pipes con nombre

Para comunicar procesos que no tienen una relación padre-hijo, tenemos *pipes* que tienen un nombre en el sistema de ficheros, se llaman *fifos*. Un *fifo* (ver *fifo(7)* y *mkfifo(3)*) es simplemente un *pipe* que se puede abrir a través de una ruta. Es un tipo de fichero especial, recuerda que hay varios tipos especiales en Unix (enlaces simbólicos, sockets, etc.).

Para crear un *fifo* en la shell, tenemos el comando `mkfifo` (ver *mkfifo(1)*). Para borrar un *fifo*, se usa `rm`. Aunque se borre el *fifo*, los procesos que lo tienen abierto podrán seguir usándolo.

Como se ve en ese ejemplo, el proceso que ejecuta `cat` lee del *fifo* los datos que escribe el proceso que ejecuta `echo`. Esos dos procesos no tienen una relación padre-hijo:

```
 $  mkfifo F
 $  echo hola >> F &
[1] 13435
 $  cat F
hola
[1]+  Done                    echo hola >> F
 $  rm F
 $
```

En un programa en C, los *fifos* se abren con `open`, como cualquier fichero. La función de biblioteca `mkfifo` crea un *fifo* con los permisos indicados en su segundo parámetro. El dueño/grupo es el UID/GID del proceso. Si el *fifo* ya existe, retorna error (-1). Para borrarlo, podemos usar la llamada al sistema `unlink`.

Tenemos que tener en cuenta estos detalles a la hora de abrir el *fifo*:

- Lectura: un `open` de solo-lectura, si no hay ningún proceso con el *fifo* abierto para escribir en él, deja al proceso bloqueado hasta que lo haya.

  Se puede forzar una apertura no bloqueante (`NON-BLOCKING`). Un `open` **no bloqueante** de solo-lectura falla si no hay ningún proceso que tenga abierto el *fifo* para escribir.

- Escritura: un `open` de solo-escritura, si no hay ningún proceso con el *fifo* abierto para leer de él, deja el proceso bloqueado hasta que lo haya.

  Si la apertura es no bloqueante (`NON-BLOCKING`), fallará si el *fifo* no está actualmente abierto para lectura.

- Se puede abrir en modo lectura-escritura (aunque en general esto no tenga mucho sentido).

- Si el *fifo* se borra mientras el proceso está bloqueado en la apertura, se queda bloqueado de por vida.

Los *fifos* son *pipes*, y se comportan de la misma forma en las lecturas y escrituras.

Si un programa quiere leer de un *fifo* todo lo que escriben distintos programas que van abriéndolo y cerrándolo, necesita reabrirlo. Veamos un ejemplo. El siguiente programa crea un *fifo* en `/tmp/logger` (si existe lo borra y lo vuelve a crear) para leer eventos del *fifo* y se queda esperando a que otros procesos abran. El programa lee líneas del *fifo* y las va escribiendo por su salida estándar junto con la hora:

Programa 6.5: logger.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <err.h>
#include <dirent.h>
#include <time.h>
#include <signal.h>
#include <fcntl.h>

enum {
    Maxline = 500,
};

void
printevent(char *e)
{
    time_t now;
    char *p;

    time(&now);
    p = ctime(&now);
    p[strlen(p) - 1] = '\0';
    printf("%s: %s", p, e);
}

int
main(int argc, char **argv)
{
    FILE *st;
    char line[Maxline];

    if (access("/tmp/logger", F_OK) == 0)
        if (unlink("/tmp/logger") < 0)
            err(EXIT_FAILURE, "cant remove /tmp/logger");

    if (mkfifo("/tmp/logger", 0664) < 0)
        err(EXIT_FAILURE, "cannot make fifo /tmp/logger");

    for (;;) {
        printevent("waiting for clients\n");
        st = fopen("/tmp/logger", "r");
        if (st == NULL) {
            errx(EXIT_FAILURE, "fopen error");
        }
        printevent("ready to read events\n");
        while (fgets(line, Maxline, st) != NULL) {
            printevent(line);
        }
        if (ferror(st)) {
            errx(EXIT_FAILURE, "read error");
        }
        printevent("client is gone\n");
        fclose(st);
    }
}
```

Si ejecutamos el programa y en otro terminal ejecutamos estos comandos:

```
$ seq 1 5 >> /tmp/logger &
$ seq 1 5 >> /tmp/logger &
$ sleep 4
$ echo hola >> /tmp/logger
```

La salida del programa `logger` podría ser:

```
$./logger
Fri Mar 27 12:41:27 2020: waiting for clients
Fri Mar 27 12:41:55 2020: ready to read events
Fri Mar 27 12:41:55 2020: 1
Fri Mar 27 12:41:55 2020: 2
Fri Mar 27 12:41:55 2020: 3
Fri Mar 27 12:41:55 2020: 4
Fri Mar 27 12:41:55 2020: 5
Fri Mar 27 12:41:55 2020: 1
Fri Mar 27 12:41:55 2020: 2
Fri Mar 27 12:41:55 2020: 3
Fri Mar 27 12:41:55 2020: 4
Fri Mar 27 12:41:55 2020: 5
Fri Mar 27 12:41:55 2020: client is gone
Fri Mar 27 12:41:55 2020: waiting for clients
Fri Mar 27 12:41:59 2020: ready to read events
Fri Mar 27 12:41:59 2020: hola
Fri Mar 27 12:41:59 2020: client is gone
Fri Mar 27 12:41:59 2020: waiting for clients
```

6.3. Sockets

Los *sockets* (ver *socket(7)*) sirven para comunicar procesos dentro del mismo sistema o a través de la red. Existen distintos tipos de sockets. Por ejemplo, los *Unix domain sockets*, *unix(7)*, sirven para comunicar procesos que ejecutan en el mismo sistema, como también hacen los *pipes* y los *fifos*. Los *network sockets* (o *internet sockets*), por ejemplo *tcp(7)* y *udp(7)*, son más interesantes, ya que nos permiten comunicar procesos que ejecutan en distintas máquinas (o en la misma máquina) a través de la red. Nos centraremos en los segundos.

En general, podemos tener *sockets* orientados a datagramas, a *streams* o a secuencias de paquetes. Dentro los *internet sockets*, los primeros usan el protocolo de transporte

UDP y los segundos TCP. Veamos como ejemplo los *sockets TCP* (orientados a *streams* de bytes).

Es común leer o escuchar que los *sockets TCP* son muy difíciles de usar, que son magia negra. Bueno, tenemos una buena noticia para ti: si has aprendido bien el uso de los ficheros y los *pipes*, **ya sabes usar los *sockets TCP***. A mucha gente le resulta muy complicado usar los *sockets TCP* porque no sabe usar bien los ficheros, no entiende la semántica de las operaciones `read` y `write` ni conoce los problemas que hemos tratado anteriormente: lecturas cortas, bloqueos, conservación de límites de escritura, etc.

Una vez configurado un *socket TCP*, se usa exactamente como un *pipe*. La única diferencia es que los *sockets* son *full-duplex*. Como en el caso de los *pipes* de Unix, no se conservan los límites de escritura. También, como los pipes, las lecturas y las escrituras se pueden bloquear. Hay que tener en cuenta que ahora tenemos *buffers* que se pueden llenar, tanto en la máquina local como en la remota.

Cuando se establece la conexión TCP, tendremos un descriptor de fichero del que podremos leer (recibir) y escribir (enviar). Como ya sabemos, esto se hace con las llamadas `read` y `write`. Hay otras operaciones para trabajar con los *sockets*, llamadas *recv(2)* y *send(2)*, que también sirven para leer y escribir. Son similares a `read` y `write`, pero aceptan algunas opciones (*flags*) para modificar el comportamiento (por ejemplo, para hacer lecturas no bloqueantes, etc.). Por lo general, nos podemos apañar con `read` y `write`. Cuando queramos cerrar la conexión, podremos hacerlo con `close`. Cuando el otro extremo cierre la conexión, leeremos cero bytes. Todo esto nos parece natural cuando ya conocemos los *pipes*. Lo único que no sabemos hacer ahora mismo es crear el *socket TCP* y establecer la conexión entre un cliente y un servidor.

En el servidor, hay que usar las funciones *socket(2)*, *bind(2)*, *listen(2)* y *accept(2)*. A continuación se muestra un ejemplo de un programa servidor que se *ata* al puerto 9999 de TCP de cualquier interfaz de red de la máquina y se queda *escuchando*, esperando a que se le conecte un cliente. Cuando pase eso, ya tendrá un descriptor de fichero para trabajar. El programa leerá del socket hasta que se le cierre la conexión, escribiendo en su salida estándar todo lo que reciba:

Programa 6.6: server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int
main(int argc, char *argv[])
{
    int fd, sockfd;
    struct sockaddr_in sin;
    struct sockaddr sclient;
    socklen_t addrlen;
    char buf[1024];
    int nr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd < 0) {
        err(EXIT_FAILURE, "socket failed");
    }
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(9999);
    if(bind(sockfd, (struct sockaddr *)&sin, sizeof(sin)) < 0){
        err(EXIT_FAILURE, "bind failed");
    }
    if(listen(sockfd, 100) < 0){ //como mucho 100 clientes en la cola
        err(EXIT_FAILURE, "listen failed");
    }

    addrlen = sizeof(sclient);
    fd = accept(sockfd, &sclient, &addrlen);
    if(sockfd < 0){
        err(EXIT_FAILURE, "accept failed");
    }
    while((nr = read(fd, buf, 1024)) > 0){
        if(write(1, buf, nr) != nr){
            err(EXIT_FAILURE, "write failed");
        }
    }
    if(nr < 0){
        err(EXIT_FAILURE, "read failed");
    }
    close(fd);
    close(sockfd);
    exit(EXIT_SUCCESS);
}
```

Aquí tenemos la implementación del cliente. Para crear el socket y conectarse, tiene que usar las funciones *socket(2)* y *connect(2)*. El cliente se conecta a la dirección 127.0.0.1

(la dirección de `localhost`) y escribe en la conexión todo lo que lee por su entrada estándar:

Programa 6.7: client.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <err.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10 #include <sys/socket.h>
11 #include <netinet/in.h>
12 #include <arpa/inet.h>
13
14 int
15 main(int argc, char *argv[])
16 {
17     struct sockaddr_in sin;
18     int fd;
19     char buf[512];
20     int nr;
21
22     fd = socket(AF_INET, SOCK_STREAM, 0);
23     if(fd < 0) {
24         err(EXIT_FAILURE, "socket failed");
25     }
26     sin.sin_family = AF_INET;
27     sin.sin_addr.s_addr = inet_addr("127.0.0.1");
28     sin.sin_port = htons(9999);
29     if(connect(fd, (struct sockaddr *)&sin, sizeof(struct sockaddr)) == -1){
30         err(EXIT_FAILURE, "connect failed");
31     }
32     while((nr = read(0, buf, 512)) > 0){
33         if(write(fd, buf, nr) != nr) {
34             err(EXIT_FAILURE, "write failed");
35         }
36     }
37     if(nr < 0){
38         err(EXIT_FAILURE, "read failed");
39     }
40     close(fd);
41     exit(EXIT_SUCCESS);
42 }
```

Se usan otras funciones como *htons(3)* para convertir los enteros al formato de red (el orden de los bytes), *inet_addr(3)* para crear una dirección IP a partir de una cadena de caracteres, etc. Si ejecutamos en un terminal:

```
 $ ls /home/
esoriano
lost+found
paurea
guest
 $ ls /home/ | ./client
 $
```

Y en otro terminal tenemos ejecutando el servidor:

```
 $ ./server
esoriano
lost+found
paurea
guest
 $
```

Vemos que todo lo que lee el cliente de su entrada estándar se escribe en el *socket*, y el servidor lo escribe en su salida estándar. El tamaño del *buffer* es distinto en ambos programas a propósito. Ahora mismo, los dos procesos están en el mismo sistema. Por supuesto, estos dos programas pueden ejecutar en dos máquinas distintas conectadas a la red.

Como podemos ver, simplemente se trata de leer y escribir en un fichero. La complicación puede estar en el uso de *socket(2)*, *bind(2)*, *listen(2)*, *accept(2)* y *connect(2)*. Para más detalles, lee su página de manual y acude a referencias como [19].

6.4. Señales

Las señales son otro mecanismo de comunicación entre procesos. Están pensadas para notificar cosas a los procesos, no para intercambiar datos. Su diseño está inspirado en el de las interrupciones hardware (y tienen problemas similares).

6.4.1. Espera activa vs. interrupciones

En general, es difícil tratar con las señales. Durante un tiempo fueron poco fiables (hasta su estandarización en el estándar POSIX.1) [19, Chapter 10] y siempre han sido complicadas de usar y entender. Sin embargo se siguen usando y son fundamentales. ¿Por qué? Por la misma razón que las interrupciones hardware. Cuando es necesario recibir una notificación, hay compromisos entre latencia, *throughput* y consumo de CPU, en el que las señales representan un papel fundamental, el extremo de notificación instantánea.

En el extremo contrario de estos compromisos está el *polling*, o *espera activa*. En el *polling*, el proceso llama a un procedimiento para comprobar si una condición ha sucedido. El proceso se meterá en un bucle llamando periódicamente a dicho procedimiento

hasta que suceda la condición[7]. Esta condición puede ser, por ejemplo, un *timeout*, que ciertos datos estén disponibles o si el programa debe acabar.

Si el proceso llama al procedimiento en un bucle sin hacer nada más, se minimiza la latencia y se aumenta el *throughput*, pero el uso de CPU se dispara (y si hay un solo procesador, puede aumentar la latencia).

Si el proceso se queda dormido durante un periodo temporal (llamando a `sleep` o `nanosleep`), disminuye el uso de CPU (y el consumo de energía) pero, a cambio, aumenta la latencia entre que sucede la condición y se recibe la notificación.

La solución a este problema es utilizar el modelo de la interrupción: el proceso se bloquea hasta que que se recibe la notificación (de desbloquear el proceso se ocupa el planificador del sistema operativo). Este es el modelo de las interrupciones y, en algunos casos es ineludible. Sin embargo, las señales no son los únicos mecanismos de sincronización que utilizan este modelo de espera bloqueada. Hay otros, alguno de los cuales hemos visto (los *pipes*) que también lo utilizan. La diferencia entre, por ejemplo, los *pipes* y las señales, reside en cuándo se realiza la notificación. Este *cuándo* hace que las señales más potentes, más peligrosas y más difíciles de utilizar: la notificación no se realiza a consecuencia de una llamada explícita del proceso (por ejemplo `read` o `write` en el caso de los *pipes*). Por eso, el proceso puede no estar preparado para recibirla o interrumpir lo que estaba haciendo. Esto nos rompe el flujo de ejecución natural de nuestros programas, que se tiene que desviar para atender la notificación.

Otra razón importante para utilizar señales es justamente interrumpir una llamada al sistema que está bloqueada (por ejemplo un read en un *pipe*).

Desde el punto de vista del proceso, una señal es similar a una interrupción hardware. Le llega una notificación en cualquier momento y tiene que manejarla.

6.4.2. Señales en Unix

Las señales pueden ser:

- **Síncronas**: se entregan inmediatamente porque son consecuencia de la ejecución del programa. En este caso, el sistema envía la señal al proceso como reacción al algo que ha hecho (normalmente mal).

  Por ejemplo, cuando se escribe en un *pipe* y no hay ningún proceso que pueda leer de él, el proceso recibe una señal (`SIGSEG`).

- **Asíncronas**: la señal se entregará y manejará cuando el proceso sea planificado. Suelen ser enviadas por otros procesos.

  Por ejemplo, un proceso puede enviar a otro proceso una señal para interrumpir su ejecución (`SIGINT`).

Cuando un proceso recibe una señal, puede ignorarla o manejarla. Para manejarla, se puede realizar la acción por omisión o se puede definir un manejador (*handler*). Hay distintas acciones por omisión dependiendo del tipo de señal:

[7] Es el mismo problema que veremos más adelante con *spin-locks* en la sección 7.3

- Terminar el proceso.
- Provocar un *core dump* y terminar el proceso[8].
- Ignorar la señal.
- Suspender su ejecución.
- Reanudar su ejecución.

Existen distintas señales, en la página de manual *signal(7)* podemos verlas todas. Cada señal tiene un nombre y un número asociado. Las más importantes son:

- Número 1, `SIGHUP`. Acción por omisión: terminar el proceso. Descripción: se ha perdido la conexión con el terminal (*line hangup*).
- Número 2, `SIGINT`. Acción por omisión: terminar el proceso. Descripción: interrumpir el programa (`Ctrl+c` en el terminal).
- Número 3, `SIGQUIT`. Acción por omisión: crear un core. Descripción: salir del programa.
- Número 4, `SIGILL`. Acción por omisión: crear un core. Descripción: instrucción ilegal.
- Número 8: `SIGFPE`. Acción por omisión: crear un core. Descripción: excepción de coma flotante.
- Número 9, `SIGKILL`. Acción por omisión: terminar el proceso. Descripción: matar el programa. No se puede ignorar ni manejar.
- Número 11, `SIGSEGV`. Acción por omisión: crear core. Descripción: violación de segmento, dirección sin traducción.
- Número 13, `SIGPIPE`. Acción por omisión: terminar el proceso. Descripción: escritura en un pipe sin lector.
- Número 14, `SIGALRM`. Acción por omisión: terminar el proceso. Descripción: alarma de *timer*.
- Número 15, `SIGTERM`. Acción por omisión: terminar el proceso. Descripción: terminación (p. ej. *shutdown*).
- Número 17, `SIGSTOP`. Acción por omisión: parar el proceso. Descripción: se quiere pausar la ejecución. No se puede ignorar ni manejar.

[8] Un *core dump* significa que se vuelca en contenido de la memoria virtual del proceso en un fichero, llamado *core*, para su inspección posterior. Dadas las condiciones para un *core dump*, que realmente se cree un fichero con el *core* o no, o dónde se genera el fichero, dependerá de la configuración del sistema. Ver *core(5)* para más detalles.

- Número 18, `SIGTSTP`. Acción por omisión: parar el proceso. Descripción: señal de parada generada por el teclado (`Ctrl+z`).
- Número 19, `SIGCONT`. Acción por omisión: descartar, el proceso se reanuda. Descripción: continuar con la ejecución.
- Número 20, `SIGCHLD`. Acción por omisión: descartar. Descripción: el estado de un proceso hijo ha cambiado.
- Número: 21, `SIGTTIN`. Acción por omisión: parar el proceso. Descripción: el proceso está en segundo plano y ha intentado leer del terminal.
- Número 30, `IGUSR1`. Acción por omisión: terminar el proceso. Descripción: señal definida por el usuario (uso libre).

6.4.3. Envío de señales

Desde la shell, podemos enviar una señal a un proceso con el comando `kill`. A este comando se le puede indicar el número de la señal o su nombre, seguido de los PID de los procesos a los que se quiere enviar. Por ejemplo, así podemos enviar una señal `SIGKILL` a un proceso:

```
$ ps
  PID TTY          TIME CMD
 4635 pts/2    00:00:00 bash
29397 pts/2    00:00:00 sleep
29398 pts/2    00:00:00 ps
$ kill -9 29397
$ ps
  PID TTY          TIME CMD
 4635 pts/2    00:00:00 bash
29402 pts/2    00:00:00 ps
[1]+  Killed                 sleep 100
$
```

Hay otros comandos para mandar señales. El comando `killall` envía una señal a los procesos fijándose en su nombre. El comando `pkill` hace lo mismo, pero aplicando un patrón (expresión regular).

También podemos enviar señales a los procesos desde el terminal, con ciertas combinaciones de teclas. Por ejemplo, `Ctrl+c` para interrumpir el programa. En el tema dedicado a los procesos ya se habló del control de trabajos (*job control*). Veamos un poco más:

- Toda sesión tiene un terminal controlador (TTY). En ocasiones, no queremos tener un TTY controlador. Por ejemplo, para crear un *demonio* (un servicio en Unix), entre otras cosas, debemos crear una sesión nueva y dejar el proceso sin terminal controlador, para que se quede ejecutando en segundo plano. Si queremos crear

un demonio, lo mejor es usar la función `daemon` de la biblioteca estándar (ver *daemon(3)*).

Para poder dejar ejecutando programas cuando cerramos su terminal, podemos usar el comando `nohup`. Con ese comando podemos ejecutar otro comando que ignore la señal `SIGHUP` (ver *nohup(1)*). Para ejecutar una sesión interactiva que no muera al cerrar el terminal, ver *screen(1)*.

- Una sesión tiene distintos grupos de procesos. La sesión está liderada por un proceso. El SID (id de la sesión) es el PID del *proceso líder de la sesión* (normalmente, la shell).
- Un grupo de procesos está liderado por un proceso. Un grupo de procesos tiene un id, el PGID, que es el PID del *proceso líder del grupo*. Un *job* es un grupo de procesos (por ejemplo, un *pipeline*).
- Como mucho, un grupo de procesos de la sesión está en primer plano (*foreground*) en el terminal.
- Al pulsar `Ctrl+c`, se envía las señal `SIGINT` a todos los procesos del grupo que está en primer plano. Al pulsar `Ctrl+z`, se envía una señal `SIGTSPT` a todos los procesos del grupo que está en primer plano.
- Si un proceso de un grupo que no está en *foreground* intenta leer del terminal, se le manda una señal `SIGTTIN` y se quedará parado (acción por omisión).

Por ejemplo:

```
 $  sleep 1000 | cat | wc &
[1] 6308
 $ jobs
[1]+  Running                 sleep 1000 | cat | wc &
 $ ps  j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 4381  4391  4391  4391 pts/0     6310 Ss    1000   0:00 /bin/bash
 4391  6306  6306  4391 pts/0     6310 S     1000   0:00 sleep 1000
 4391  6307  6306  4391 pts/0     6310 S     1000   0:00 cat
 4391  6308  6306  4391 pts/0     6310 S     1000   0:00 wc
 4391  6310  6310  4391 pts/0     6310 R+    1000   0:00 ps j
  $
```

Podemos observar que hay una sesión ligada al TTY `pts/0` con SID 4391, que está liderada por el proceso que ejecuta la shell (`bash`). El grupo en primer plano es el que lidera el PID 6310, los procesos 6306, 6307 y 6308 pertenecen a otro grupo con PGID 6306 que está en segundo plano (los procesos creados para el *pipeline*). Si en este momento (mientras que se ejecuta el `ps`) se pulsara `Ctrl+z` en el terminal, se enviaría el `SIGTSPT` al proceso 6310.

Tanto el grupo de procesos como la sesión se heredan del padre y se conservan después de un `exec`. La llamada al sistema *setpgid(2)* sirve para cambiar el grupo de procesos. La llamada al sistema *setsid(2)* crea una nueva sesión, siendo el proceso llamador el líder.

Para enviar señales desde un programa, tenemos la llamada al sistema *kill(2)*:

```
int  kill(pid_t pid, int sig);
```

Para poder enviar una señal, el UID del proceso destino debe ser el mismo que el del proceso (o ser root). La única excepción es `SIGCONT` (pero el proceso deber ser descendiente). Si el primer parámetro es 0, se pone la señal a todos los procesos del grupo de procesos al que pertenece el proceso que hace la llamada. Si es -1, manda la señal a todos los procesos que se pueda menos a los del sistema y él mismo.

La función *killpg(3)* es similar, pero manda la señal a un grupo de procesos:

```
int  killpg(pid_t pgrp, int sig);
```

El primer parámetro es el PGID del grupo al que se le quiere enviar la señal.

Si el valor de `sig` es 0, no se manda ninguna señal, sino que sencillamente se comprueba que el identificador del grupo (`gid`) o del proceso (`pid`) existe (es una forma de comprobar esto).

6.4.4. Manejo de señales

Un proceso puede tener una señal **bloqueada**. Si es así, cuando le envíen esa señal no se entregará. La señal se queda *pendiente de entregar* (se entregará cuando se desbloquee la señal). Esto es similar a la máscara de interrupciones de un procesador.

Si la señal **no está bloqueada**, puede pasar una de estas cosas:

- Que se realice la acción por omisión para dicha señal (ver en la página de manual *signal(7)*).
- Que se realice una acción especial definida por un manejador de señal, que ha debido ser instalado previamente.
- Que se ignore la señal. No es lo mismo ignorar una señal que bloquearla. Al ignorarla se pierde, al bloquearla, se aplaza.

  ¿Recuerdas los procesos *zombie*? Para evitar dejar procesos zombie, un proceso tiene que ignorar la señal `SIGCHLD` para informar al sistema de que no está interesado en el estado de salida de sus hijos y no va a hacer `wait` por ellos. Tras ignorar esa señal, los hijos no se quedarán *zombie*.

En Linux podemos asignar una acción para una señal con `signal`:

```
typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
```

Observa que lo primero es una definición de *puntero a función* en C. Significa que, tanto lo que se le pasa a `signal` como lo que devuelve, son direcciones que apuntan al *texto* del programa (instrucciones). Esos punteros son la forma de pasar como argumento (y devolver como resultado) manejadores para las señales.

Se puede registrar un manejador para una señal, poner su acción por omisión (pasando la constante `SIG_DFL`) o marcar la señal como ignorada (pasando la constante `SIG_IGN`). Se sobrescribe el estado anterior. Los manejadores de señal no retornan nada y tienen un único parámetro de tipo entero (el número de la señal).

Esta función es una versión simplificada de la llamada al sistema `sigaction`. El problema es que `sigaction` es muy complicada de usar (ver *sigaction(2)*). Sin embargo, `signal` no es estándar [19, Chapter 10]: originalmente era una llamada existente en los sistemas de la familia Unix System V (los sistemas basados en BSD incorporaron una función después), no es igual en los sistemas en los que existe y además ha cambiado con el tiempo en las distintas versiones del Linux. Usaremos `signal` por simplicidad, pero hay que tener en cuenta todo esto si se desea hacer programas portables (en ese caso debes usar `sigaction`).

Veamos un ejemplo. El siguiente programa maneja las señales `SIGINT` y `SIGTERM`. También intenta ignorar `SIGKILL`:

Programa 6.8: ignoresig.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>

void
handler(int number)
{
    switch (number) {
    case SIGINT:
        fprintf(stderr, "SIGINT received!\n");
        break;
    case SIGTERM:
        fprintf(stderr, "SIGTERM received!\n");
        break;
    default:
        fprintf(stderr, "other: %d\n", number);
    }
}

int
main(int argc, char *argv[])
{
    signal(SIGINT, handler);
    signal(SIGTERM, handler);
    // this won't work... you cannot ignore sigkill
    signal(SIGKILL, SIG_IGN);

    for (;;) {
        sleep(1);
        printf("I am here! my PID is: %d\n", getpid());
    }
}
```

Si ejecutamos el programa e intentamos acabar su ejecución pulsando `Ctrl+c` en el terminal, veremos que se recibe la señal. Como la estamos manejando, no se realiza la acción por omisión (terminar la ejecución). Si después, desde otro terminal, le mandamos una señal `SIGKILL`, veremos que no se ignora y el proceso muere. La señal `SIGKILL` no se puede ignorar:

```
 $ ./ignoresig
I am here! my PID is: 8263
 ^C
SIGINT received!
I am here! my PID is: 8263
I am here! my PID is: 8263
I am here! my PID is: 8263
I am here! my PID is: 8263
Killed
 $
```

Hay que tener en cuenta que cuando se ignora o maneja una señal síncrona, se volverá a intentar ejecutar la instrucción que la ha provocado. Por ejemplo, si una instrucción provoca un `SIGSEGV` y manejamos dicha señal, al retornar se intentará ejecutar de nuevo la instrucción. Si el manejador no ha hecho nada por evitar el error, volverá a recibir la señal (y a ejecutarse el manejador), entrando en un bucle infinito. Esto también puede ocurrir si ignoramos la señal. El estándar POSIX indica que el comportamiento cuando se ignoran ciertas señales como `SIGSEGV` es indeterminado (también existe el problema del bucle infinito). En Linux, si se ignora `SIGSEGV`, al intentar ejecutar la instrucción de nuevo volverá a recibirse la señal y el programa morirá con un *segmentation fault*.

Hay que tener mucho cuidado a la hora de escribir un manejador de señal. Es importante tener claro lo que está pasando: tu programa estaba ejecutando tranquilamente y de repente se le ha interrumpido para pasar a ejecutar un manejador. Esto es uno de los factores que hace muy complicado usar señales: se rompe el flujo de control de tu programa.

En primer lugar, el manejador debería ser lo más **pequeño** posible. Además, como se pueden recibir señales mientras que se maneja una señal (ya sabemos que se pueden anidar), el manejador debe ser **reentrante**. Esto significa que se tiene que poder llamar de nuevo a esa función antes de que haya retornado la llamada actual.

Esto complica mucho las cosas. Por ejemplo, tenemos que tener mucho cuidado con las variables globales y estáticas, porque se puede llegar a estados incoherentes. ¿Qué pasa si se ejecuta el manejador mientras que estás actualizando la variable global que va a querer usar la nueva llamada? Entonces, es problemático usar variables globales o estáticas, pero hay que tener en cuenta que un manejador no retorna nada (¡tu programa no le llama!), por lo que la única forma de modificar el estado del programa (si se necesita) es mediante una variable global. Si usamos nuestras variables globales, hay que hacerlo con mucho cuidado. No podemos usar ninguna variable global que no controlemos nosotros. Por ejemplo, un manejador no debería tocar la variable `errno`.

Por el mismo motivo, el manejador tampoco puede llamar a ninguna otra función que no sea reentrante (p. ej. `malloc`, `free`, etc.). Por ejemplo, ¿qué pasa si la señal ocurrió mientras se estaba ejecutando una llamada a `malloc` y tu manejador llama a `malloc`? En ese caso, seguramente se corrompa la estructura interna del heap.

Hay una lista de funciones que se pueden usar durante el manejo de una señal. Esas funciones son reentrantes y están preparadas para poderse usar manejando una señal

(se encargan de bloquear las señales para evitar problemas, etc.). Se llaman funciones *async-signal-safe*. En Linux, están descritas en la página de manual *signal-safety(7)*.

En el último programa que hemos descrito (`ignoresig.c`), llamamos a `fprintf` desde el manejador... ¿es correcto? No, no lo es. Esa función no es *async-signal-safe*. Como ese programa es simplemente una prueba para ver el funcionamiento de la señales, lo podemos dejar así, pero podría tener un comportamiento impredecible y erróneo.

Pero eso no es todo. ¿Qué pasa si se envía una señal a un proceso que está realizando en ese momento una llamada al sistema? Las llamadas al sistema se dividen en dos tipos: lentas y rápidas. Una llamada al sistema **lenta** es aquella que puede bloquear temporalmente al proceso. Son llamadas como `open`, `read`, `write`, `wait`, etc. Si el proceso recibe una señal mientras está en una llamada al sistema lenta, la llamada al sistema falla (esto es, es *interrumpida*).

El comportamiento de las llamadas lentas se puede modificar con la función `siginterrupt`:

```
int siginterrupt(int sig, int flag);
```

Algunas llamadas al sistema lentas permiten ser reiniciadas en caso de que se les interrumpa (no todas lo soportan). En ese caso, si se recibe una señal mientras se está realizando una llamada al sistema, la llamada se reinicia (si es posible). Para activar la interrupción, se pasa la señal y el flag a 1. Para desactivar la interrupción (esto es, para que se reinicie la llamada si es posible) hay que pasar un 0. En Linux, por omisión está a 0 (se reinician las llamadas si es posible).

Veamos un ejemplo con un temporizador. La llamada al sistema `alarm` sirve para programar un *timer*: se especifica un tiempo en el que el proceso recibirá una señal de tipo `SIGALRM`:

```
unsigned int alarm(unsigned int seconds);
```

El tiempo que se le pasa en en segundos. Ten en cuenta que se recibirá la señal **aproximadamente** en ese tiempo. Si se le pasa un 0, se desactiva el temporizador. Este mecanismo se suele usar para programar *timeouts*, esto es, determinar un tiempo para que se realice una acción (normalmente leer de algún sitio, enviar o recibir un evento, etc.).

Supongamos el siguiente programa, que admite un único argumento con el valor que se le quiere pasar a `siginterrupt`. Programa un temporizador de 5 segundos y se pone a leer de la entrada estándar. Si la lecura es interrumpida por el temporizador, avisa de ello por la salida de errores y acaba. Si no, se queda bloqueado en el `read` hasta que pueda leer de la entrada y después imprime lo que ha leído:

Programa 6.9: timeout.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>

enum {
    Bsize = 1024,
    Timeout = 5,
};

void
handler(int number)
{
    // just an example, fpritnf is not async-signal-safe
    fprintf(stderr, "This is the handler, is the syscall restarted?\n");
}

int
main(int argc, char *argv[])
{
    int setflag;
    char buf[Bsize];
    int nr;

    if (argc != 2) {
        err(EXIT_FAILURE, "usage: timeout [0|1]");
    }
    setflag = atoi(argv[1]);
    if (setflag != 0 && setflag != 1) {
        err(EXIT_FAILURE, "Bad argument");
    }

    signal(SIGALRM, handler);
    siginterrupt(SIGALRM, setflag);

    alarm(Timeout);
    nr = read(0, buf, sizeof(buf) - 1);
    alarm(0);
    if (nr < 0) {
        if (errno == EINTR) {
            fprintf(stdout, "timeout!\n");
        } else {
            err(EXIT_FAILURE, "read error");
        }
    } else {
        buf[nr] = '\0';
        printf("read: %s", buf);
    }
    exit(EXIT_SUCCESS);
}
```

Si ejecutamos el programa y escribimos algo en la entrada antes de que pasen los cinco

segundos:

```
 $ ./timeout 1
hola
read: hola
 $
```

En este caso, se ha desactivado el temporizador y no ha saltado. Si ejecutamos el programa con un 0 como argumento, haría lo mismo.

Si ejecutamos el programa dejando pasar 5 segundos, saltará. Veamos que pasa con cada valor posible para el argumento. Con el reinicio desactivado:

```
 $ ./timeout 1
This is the handler, is the syscall restarted?
timeout!
 $
```

En este caso, se ha interrumpido la llamada al sistema, el *timeout* ha funcionado: después de 5 segundos esperando a que nos escriban en la entrada, hemos abortado la operación.

Si probamos ahora con el reinicio activado, vemos que salta el temporizador pero se sigue leyendo de la entrada. Después escribimos en su entrada y el programa termina:

```
 $ ./timeout 0
This is the handler, is the syscall restarted?
hola
read: hola
 $
```

6.4.5. Bloquear señales

Un programa puede bloquear las señales en las regiones de código en las que no quiere ser interrumpido. Cuando se recibe una señal que está bloqueada, esta queda **pendiente** de entregar. Por tanto, no se pierden. El programa puede comprobar qué señales están pendientes de entregar y actuar en consecuencia. Cuando se desbloquea la señal, **se entregará si está pendiente**.

El proceso tiene una **máscara de señales**[9] que indica qué señales están bloqueadas. La llamada al sistema *sigprocmask(2)* sirve para modificar o ver las señales bloqueadas. La llamada al sistema *sigpending(2)* nos da el conjunto de señales que están pendientes de entregar. Veamos un ejemplo:

[9]Similar a una máscara de interrupciones hardware.

Programa 6.10: blocksig.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <err.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10 #include <signal.h>
11
12 void
13 handler(int number)
14 {
15     // just an example, fprintf is not async-signal-safe
16     fprintf(stderr, "I am the handler\n");
17 }
18
19 void
20 printpending()
21 {
22     sigset_t pending;
23     int i;
24
25     sigemptyset(&pending);
26     sigpending(&pending);
27     fprintf(stderr, "Pending signals:");
28     for (i = 1; i < NSIG; i++) {
29         if (sigismember(&pending, i)) {
30             fprintf(stderr, " %02d", i);
31         }
32     }
33     fprintf(stderr, "\n");
34 }
35
36 int
37 main(int argc, char *argv[])
38 {
39     sigset_t set;
40
41     fprintf(stderr, "Blocking SIGINT for 10 seconds\n");
42     signal(SIGINT, handler);
43     sigemptyset(&set);
44     sigaddset(&set, SIGINT);
45     sigprocmask(SIG_BLOCK, &set, NULL);
46     sleep(10);
47
48     printpending();
49
50     fprintf(stderr, "Unblocking SIGINT\n");
51     sigprocmask(SIG_UNBLOCK, &set, NULL);
52     fprintf(stderr, "Main done, bye bye!\n");
53     exit(EXIT_SUCCESS);
54 }
```

El programa instala el manejador para la señal `SIGINT` y después la bloquea durante 10 segundos. Cuando pasan los 10 segundos, muestra todas las señales que están pendientes

de entregar. Después, desbloquea la señal `SIGINT`, escribe un mensaje de despedida y termina.

La constante `NSIG` de la biblioteca estándar contiene el número de señales del sistema. Las llamadas `sigprocmask` y `sigpending` usan el tipo de datos `sigset_t` para representar un conjunto de señales. Hay varias funciones para manejar conjuntos: `sigemptyset` inicializa un conjunto vacío, `sigaddset` añade una señal al conjunto y `sigismember` nos dice si una señal es miembro de un conjunto.

Veamos cómo se comporta. Si ejecutamos el programa y no hacemos nada durante los 10 segundos:

```
 $ ./sigmask
Blocking SIGINT for 10 seconds
Pending signals:
Unblocking SIGINT
Main done, bye bye!
 $
```

Si ahora lo volvemos a ejecutar y pulsamos `Ctrl+c` en el terminal antes de que pasen los 10 segundos:

```
 $ ./sigmask
Blocking SIGINT for 10 seconds
 ^C
Pending signals: 02
Unblocking SIGINT
I am the handler
Main done, bye bye!
 $
```

Vemos que la señal 2 (`SIGINT`) está pendiente de entregar **cuando pasan los 10 segundos**. Aunque se haya pulsado `Ctrl+c`, el programa duerme 10 segundos, ya que la señal está bloqueada. Después vemos que, al desbloquear la señal, se recibe (vemos lo que escribe el manejador) y se acaba el programa.

Todo el estado del proceso relacionado con las señales se hereda del padre. Después de un `exec` se restablecen las acciones por omisión, pero se respeta la configuración sobre señales bloqueadas.

6.4.6. Implementación

¿Y cómo funciona todo esto? El kernel mantiene la información sobre las señales (manejadores, máscara, etc.) en la estructura de datos que representa al proceso.

Grosso modo, esto es lo que ocurre cuando se envía una señal a un proceso:

1. La señal se entrega al proceso al entrar al kernel por cualquier motivo: por haber realizado una llamada al sistema, por una interrupción hardware, etc. También se entregan al salir del kernel.

2. Se inserta un *signal frame* en la pila de usuario del proceso. Esta estructura contiene el contexto actual del proceso (valor de los registros, etc)[10]. Si no se desea usar la pila para esto, se puede configurar con la función `sigstack` (ver *sigstack(3)*).

3. Se mete en la pila un contador de programa que contiene la dirección de memoria donde está el *stub* de la biblioteca estándar para la llamada al sistema `sigreturn`.

4. Se realiza la llamada a la función instalada como manejador de la señal. Su contador de programa de retorno será la que se ha metido en la pila en el paso anterior.

5. Cuando el manejador retorna, por como tenemos la pila, se retorna al stub de la llamada al sistema `sigreturn`. Por tanto, al retornar, se termina haciendo una llamada al sistema `sigreturn`.

6. El kernel ejecuta las acciones de `sigreturn`: saca el *signal frame* de la pila de usuario y restaura el contexto del proceso.

7. Cuando se vuelva a planificar el proceso, continuará donde fue interrumpido.

En resumen, **las señales son complicadas de usar y rompen el flujo de nuestros programas**. Si no es estrictamente necesario usarlas, es mejor buscar alternativas. Si tenemos que usarlas, debemos ser muy cuidadosos para evitar comportamientos erróneos. Sus usos comunes son *timeouts* e interrupciones periódicas de reloj, matar o notificar de que debe salir a un proceso en ejecución, o, en el caso de las señales síncronas, recuperarse o generar un mensaje de error en caso de un error grave (por ejemplo un *segmentation fault*).

En los programas multihilo, hay que tener todavía más cuidado con el uso de señales (veremos más sobre esto en la sección 7.5.1).

[10] Es en realidad una estructura de tipo `ucontext_t`, descrita en *getcontext(3)*.

6.5. Ejercicios no resueltos

1. Continúa la minishell de los capítulos 4 y 3. Añádele la posibilidad de ejecutar pipelines. Tienes que esperar por uno o todos los procesos del pipeline. Implementa una de las dos opciones. Mira cual usa la shell de tu sistema. ††∗∗

2. Añade `job control` a la minishell anterior. † † †∗∗

3. Implementa un programa en C en el que los hijos hagan algo que requiera un tiempo fijo. Puede ser hacer `ping` a una máquina (por ejemplo `ping -c 1`) para ver si está levantada, o esperar a algo mediante un *timeout*. Ejecútalo en un sólo proceso esperando a que acabe, y después en varios. Mide el tiempo que tarda cada uno, por ejemplo con el comando `time`.†∗

4. Implementa el programa anterior con los hijos escribiendo el resultado en un *pipe* y el padre recibiendo las respuestas. Mide de nuevo lo que tarda.†∗

5. Escribe dos programas en C, un cliente y un servidor que usen un *socket TCP*. Cada vez que se ejecute el cliente, debe listar los procesos que están ejecutando en la máquina servidora (ejecutando un `ps aux`). El servidor no debe terminar su ejecución tras responder a un cliente, debe esperar a otros clientes. †∗

6. Instala el paquete `wamerican`. Como ya vimos, este paquete incluye un fichero con palabras `/usr/share/dict/words`. Escribe un programa en C que lea palabras de la entrada estándar y si están en el fichero, las escriba en mayúsculas. Para ello llamará al comando `fgrep` (mediante `fork` y `exec`). Escribe una versión que utilize `tr` y una en la que el padre las pase a mayúsculas en su código.†∗

7. Escribe un programa en C que cuando reciba una señal `TERM`, escriba su estado y continúe ejecutando. Ten en cuenta que esto no significa que haya que escribir el estado en el manejador de la señal. El programa puede escribir la hora cada segundo, por ejemplo. Como estado, puede imprimir cuanto tiempo lleva ejecutando o cuantas iteraciones.†∗

8. Escribe un programa que haga timeout esperando en un `fifo` un valor.†∗

9. Escríbete un pequeño planificador colaborativo en C para threads de espacio de usuario que permita ejecutar varios threads sobre un proceso. Cada thread ejecutará una función main con una signatura concreta, por ejemplo `void main(void *v);`. Puede utilizar las funciones *getcontext(3)*, *setcontext(3)*, *makecontext(3)* y *swapcontext(3)* para crear, salvar y recuperar el contexto de un thread. Implementa al menos `yield` para ceder el proceso a otro thread. † † †∗ ∗ ∗

10. Reescribe el planificador anterior para que sea expulsivo utilizando *alarm(2)*. Recuerda configurar las llamadas al sistema para que rearranquen y se cuidadoso con no utilizar ninguna función de biblioteca que utilice `SIGALARM` (y por tanto interfiera con el planificador). Puedes implementar tu propio `sleep` que colabore con

el planificador (en lugar del que implementa el sistema, que seguramente utilice `SIGALARM`.† † †∗ ∗ ∗

7 Introducción a la programación concurrente

Este capítulo es una introducción a la programación concurrente con flujos de ejecución que comparten memoria (o, más en general, comparten recursos). Ya han aparecido problemas de concurrencia en los capítulos previos. Por ejemplo, cuando un proceso crea otro proceso, ¿quién ejecuta antes? ¿padre o hijo? Ya vimos en su momento que no hay garantías sobre eso: los dos son procesos concurrentes que pueden ordenar su ejecución de cualquier forma. Sólo podemos estar seguros si forzamos un orden usando algún mecanismo de sincronización. En esta introducción estudiaremos la primitiva *cerrojo* (*lock*).

Vamos a ver la implementación en detalle de una de estas primitivas en el sistema operativo Plan 9. Al ser especialmente sencilla, se pueden entender todos los detalles implicados. A continuación veremos cómo se utilizan estas primitivas en Unix tanto para compartir memoria como para compartir ficheros.

7.1. Memoria compartida y condiciones de carrera

Una **condición de carrera** se produce cuando el resultado del programa depende de cómo se ordenen los distintos flujos de ejecución concurrentes, y no se sabe cómo lo harán. Esto puede suceder siempre que hay más de un flujo de ejecución accediendo a recursos compartidos. Los flujos no tienen por qué estar ejecutando en la misma máquina y los recursos compartidos pueden ser bases de datos, sensores o actuadores de un robot, ficheros en un sistema distribuido, etc.

Este problema se complica si entre los recursos que se comparten hay **memoria compartida**: varios flujos de ejecución leen/escriben variables compartidas concurrentemente[1]. Esto es, los procesos (i.e. flujos) comparten sus segmentos DATA y BSS. Cuando uno de ellos escribe una variable compartida, el resto de flujos ven la modificación.

Una condición de carrera es el peor tipo de *bug* posible. En general, una condición de carrera no es reproducible. Esto quiere decir que el fallo ocurre con cierta probabilidad, (que puede ser muy baja) lo que dificulta su descubrimiento y depuración. El primer paso para arreglar un *bug* es encontrar una forma sistemática de reproducirlo. En este caso puede ser imposible o muy difícil (puede costar mucho tiempo e intentos cada vez que se intenta reproducir). También puede ser muy difícil comprobar si se ha arreglado el problema tras introducir una modificación.

Los resultados de una condición de carrera pueden ser sorprendentes e ininteligibles. Sólo una lectura detallada y la comprensión exhaustiva del código nos puede dar alguna garantía de que no hay condiciones de carrera[2].

[1] Esta complejidad es consecuencia de la sofisticación de la jerarquía de memoria de un sistema multicore moderno, con varios niveles de *caché* compartida, relajaciones de las garantías en el orden de acceso a la memoria y todo tipo de optimizaciones (ver [28, Capítulo 5]). Un análisis detallado y en profundidad de los diferentes modelos de memoria tanto hardware como software se pueden ver en [29].

[2] Hay algunas otras herramientas que pueden ayudar a librar nuestro código de condiciones de carrera. Las dos más comunes con: los detectores de condiciones de carrera en ejecución (como, por ejemplo el que tiene el *runtime* de Go[30]) y los sistemas de especificación formal como TLA^+[31]. Los detectores de carreras no detectan todas, pero pueden ayudar. Los sistemas de especificación formal son limitados y es complicado establecer la relación entre el modelo y el código o sistema subyaciente. Sin embargo,

7.1.1. Ejemplo 0

Si una operación sobre el recurso compartido no es **atómica**, dos o más flujos pueden interferir entre ellos haciendo que el resultado de las operaciones sea incorrecto. Una **operación atómica** es una operación indivisible: ninguna otra operación que ejecute otro flujo puede interferir con ella. Dicho de otra forma, *la operación se hace de una tacada*, de tal forma que ningún otro flujo de control puede interferir mientras se ejecuta. En general, pocas operaciones son atómicas.

Una definición más cuidadosa de **operación atómica** se puede dar en términos de estados posibles del sistema. Si la operación es atómica, el sistema entero puede estar en un estado anterior a la realización de la operación o en un estado posterior, pero no en un estado intermedio[3]. Es similar al concepto de transacción en una base de datos.

Supongamos que las líneas del siguiente programa sí son atómicas (no es así en realidad, pero simplifica el ejemplo). La variable `x` (inicializada a 0) es una variable compartida entre dos flujos de control, que llamaremos A y B. Las variables `aux` e `i` son locales y no se comparten, cada flujo tiene su propia copia.

Los flujos A y B ejecutan este código concurrentemente:

```
1 int i = 0, aux = 0;
2 for(i=0; i<10; i++){
3     aux = x;
4     aux = aux + 1;
5     x = aux;
6 }
7 // value of x?
```

¿Cuáles son los posibles valores finales de la variable x? ¿Cuál es el máximo que puede alcanzar? ¿Y el mínimo?

El máximo que puede alcanzar es 20, si el proceso A ejecuta todas sus iteraciones y después el proceso B ejecuta todas sus iteraciones, el contador se incrementa 20 veces. Esto es fácil de ver.

Es más complicado ver que hay incrementos que se puede perder: esto sucede si A se encuentra en la línea 4, después ejecuta B desde la línea 3 a la 5, y luego A ejecuta la línea 5. En este caso, el incremento realizado por B se pierde.

Podríamos pensar que, si se perdieran todos los incrementos de un flujo de control (por ejemplo los de B), entonces el mínimo valor de la variable x al acabar sería 10. No es así: el mínimo puede ser menor que 10.

Imagina la siguiente secuencia:

pueden ayudar para depurar la concurrencia de un algoritmo. La comprobación automática de la corrección de algoritmos y programas concurrentes es un espacio fértil de investigación y se están produciendo grandes avances en él. De momento, aunque estas herramientas pueden ayudar, nada sustituye una lectura cuidadosa del código.

[3] Esta definición ligeramente diferente es más fácil de generalizar cuando tengo un jerarquía de memoria con cachés a varios niveles y varios procesadores con cachés locales y registros que comparten memoria. Requiere que los efectos en los diferentes niveles de caché de la operación se sincronicen antes de completarla sin que los otros procesadores vean resultados intermedios.

1. El flujo A ejecuta hasta la línea 3:

```
1 int i = 0, aux = 0;      ← B(x=0,aux=0,i=0)
2 for(i=0; i<10; i++){
3     aux = x;             ← A(x=0,aux=0,i=0)
4     aux = aux + 1;
5     x = aux;
6 }
7 // value of x?
```

2. El flujo B ejecuta 9 iteraciones y se queda en la línea 2:

```
1 int i = 0, aux = 0;
2 for(i=0; i<10; i++){     ← B(x=9,aux=9,i=9)
3     aux = x;             ← A(x=9,aux=0,i=0)
4     aux = aux + 1;
5     x = aux;
6 }
7 // value of x?
```

3. El flujo A ejecuta su primera vuelta, se queda en la línea 2 y deja `x` a 1:

```
1 int i = 0, aux = 0;
2 for(i=0; i<10; i++){     ← B(x=1,aux=9,i=9), A(x=1,aux=1,i=1)
3     aux = x;
4     aux = aux + 1;
5     x = aux;
6 }
7 // value of x?
```

4. El flujo B avanza hasta la línea 3, su `aux` queda con valor 1:

```
1 int i = 0, aux = 0;
2 for(i=0; i<10; i++){     ← A(x=1,aux=1,i=1)
3     aux = x;             ← B(x=1,aux=1,i=9)
4     aux = aux + 1;
5     x = aux;
6 }
7 // value of x?
```

5. El flujo A ejecuta todas sus vueltas:

```
1 int i = 0, aux = 0;
2 for(i=0; i<10; i++{
3     aux = x;              ← B(x=9,aux=1,i=9)
4     aux = aux + 1;
5     x = aux;
6 }
7 // value of x?           ← A(x=9,aux=9,i=10)
```

6. El flujo B termina su última iteración:

```
1 int i = 0, aux = 0;
2 for(i=0; i<10; i++){
3     aux = x;
4     aux = aux + 1;
5     x = aux;
6 }
7 // value of x?           ← A(x=2,aux=9,i=10), B(x=2,aux=2,i=10)
```

Por tanto, **el mínimo puede ser 2**. Sorprendente, ¿verdad?

7.1.2. Ejemplo 1

Veamos otro ejemplo. Supongamos otra vez dos flujos de ejecución, A y B, que comparten la variable `x` inicializada a 0. Los dos flujos ejecutan estas sentencias concurrentemente:

```
1 x++;
2 printf("%d", x);
```

Ahora ya no suponemos que cada una de esas sentencias son atómicas. Los posibles resultados son los siguientes:

- A imprime 1 y B imprime 2.
- B imprime 1 y A imprime 2.
- A y B imprimen 2 (A ejecuta la línea 1, B ejecuta la línea 1 , A ejecuta la línea 2 y B ejecuta la segunda 2).
- Dependiendo en las instrucciones que requiera el incremento en la arquitectura, pueden imprimirse cualquier par de enteros. Hay que tener muchas cosas en cuenta: caches, instrucciones exóticas, usar un registro como almacenamiento temporal, optimizaciones del compilador, etc.

7.1.3. Ejemplo 2

Para intentar arreglar el ejemplo 1, podríamos intentar utilizar una nueva variable compartida llamada `busy` que está inicializada a 0. La idea es que los flujos A y B usen esa variable para detectar si el otro flujo está modificando el valor de `x`, y si es así, no incrementarlo (en ese caso no imprimiría nada, pero evitaría imprimir un valor sin incrementar o valores absurdos):

```
1 if(! busy){
2     busy = 1;
3     x++;
4     printf("%d", x);
5     busy = 0;
6 }
```

¿Hemos eliminado la condición de carrera? **No**. Supongamos esta ejecución (suponiendo que cada línea se corresponde con una instrucción atómica) :

1. A evalúa la condición del *if* y es *true*. Ejecuta la línea 1 que podemos escribir[4] $[l_{A,1}]$.

2. B evalúa *"a la vez"* la condición del *if* y es *true*. Podemos escribirlo: $[l_{A,1}||l_{B,1}]$.

3. A ejecuta el cuerpo del if mientras que B ejecuta el cuerpo del *if*. Una posible ejecución es $[l_{A,1}||l_{B,1}, l_{B,2}||l_{A,2}]$. Incluso si no se ejecuta nada en paralelo pero se alternan los flujos de ejecución (concurrencia), puede haber un problema. Otra posible ejecución es: $[l_{A,1}, l_{B,1}, l_{B,2}, l_{A,2}]$, ambos flujos han entrado en el *if*.

4. Se dan los mismos resultados posibles del Ejemplo 1.

En general, este tipo de condición de carrera (que aquí se da con la variable `busy`) se denomina TOCTOU (***T**ime **o**f **C**heck, **T**ime **o**f **U**se*). Se da siempre que se comprueba el estado de un recurso y después se usa dicho recurso, si pueden ocurrir cosas que cambien las condiciones para usarlo (esto es, las condiciones se daban en el momento de la comprobación, pero no en el momento del uso).

7.1.4. Ejemplo 3

Supongamos que tenemos dos hilos. Las dos variables `done` y `x` empiezan a 0 porque son globales. Uno ejecuta:

[4] Como notación, vamos a escribir entre corchetes una secuencia de instrucciones $[i_1, i_2, i_3]$, con la coma para expresar que suceden una tras otra y dos líneas verticales $||$, es decir $[i_1||i_2||i_3]$, cuando ejecutan en paralelo. Por ejemplo $[i_1, [i_1||i_2], i_3]$, significa que ejecuta primero i_1. A continuación i_1 en paralelo a i_2 y cuando acaben ambos i_3.

```
1 while(!done){
2     ;
3 }
4 printf("%d", x);
```

y el otro:

```
1 x = 1;
2 done = 1;
```

¿Puede suceder que el segundo thread imprima 0? La respuesta es sí. Y puede suceder por varias razones. La primera por el hardware. Muchos procesadores no garantizan que los efectos en la memoria de un core se vean en el mismo orden en otro (sí en el mismo). Así, aunque el segundo thread asigne 1 a `x` y luego a `done`, el primero puede que vea las actualizaciones en el orden contrario (por la sincronización de las diferentes cachés). Pero es peor que eso. Algunos compiladores (incluyendo el de C) pueden reordenar las instrucciones si no ven que *en este trozo de código* hay dependencias de datos. No contemplan la posibilidad de que haya otro hilo ejecutando a la vez. Pueden reescribir el segundo trozo de código y generar código máquina que en realidad sea equivalente a:

```
1 done = 1;
2 x = 1;
```

Esto puede pasar, por ejemplo, cargando las variables a registros, manipulándolas y luego escribiéndolas a memoria al final de la función.

Este tipo de código, que no utiliza las primitivas de sincronización que vamos a ver más adelante, es un ejemplo de lo que en C se llama *comportamiento indefinido* (*undefined behaviour*). El compilador puede generar código sin sentido porque el código de entrada se sale de la especificación. Esto no sucede sólo con las condiciones de carrera (por ejemplo, con accesos a memoria erróneos pueden suceder cosas similares también).

El modelo de memoria de C recibe el nombre de *DRF-SC or Catch Fire* [5]. Si no hay sincronización, puede pasar **cualquier cosa** (en broma se dice que puede arder el procesador, de ahí lo de *catch fire*). Esto significa que, en ocasiones, el comportamiento de código erróneo es incomprensible. Hay que ser muy cuidadoso. En el ejemplo anterior, el optimizador podría, por ejemplo, borrar el `printf` ya que considera que no va a ejecutar nunca.

[5] ***D**ata **R**ace **F**ree **S**equential **C**onsistency*: sin condiciones de carrera y con apariencia de ejecución secuencial consistente

7.2. Exclusión mutua

Lo que estaba tratando de hacer el Ejemplo 2 sin éxito es proporcionar **exclusión mutua**: cuando un flujo está dentro de una región, no puede ejecutar simultáneamente ningún otro. Esto es, la ejecución de las regiones de código son *mutuamente excluyentes*.

Este concepto se puede definir de forma similar a como hacíamos con las operaciones atómicas. Al final, un conjunto $(P, Q, R, \ldots)$ de regiones de instrucciones (que denotaríamos $\{p_1, p_2, p_3, \ldots\}$, $\{q_1, q_2, q_3, \ldots\}$, $\{r_1, r_2, r_3, \ldots\}\ldots$) ejecutan con **exclusión mutua** entre ellos si el sistema sólo puede pasar por estados intermedios en los que han ejecutado en una secuencia concreta, por ejemplo $RPQ\ldots$ de comienzo a fin.

Por ejemplo, la ejecución $[p_1, q_1, p_2 \ldots]$ es errónea[6]. También es errónea $[p_1, [q_1||p_2] \ldots]$. En ambos casos P y Q ejecutan de forma concurrente (en el primer caso de forma concurrente o pseudoparalela y en el segundo caso con auténtico paralelismo). En cada momento del tiempo sólo uno de estos conjuntos puede encontrarse a medio completar y el resto haberse completado antes de comenzar éste (y producido todos sus efectos en los estados del sistema) o no haber comenzado todavía (y no habiendo producido ninguno de sus efectos en los estados del sistema). Observa que $[p_1, p_1, p_2] \ldots$ también sería errónea (P no puede ejecutar de forma concurrente a P[7]).

La diferencia con el concepto de atomicidad es que una instrucción ejecuta de forma atómica si tiene exclusión mutua con *todo el resto de instrucciones*. La exclusión mutua es *sólo entre las regiones de instrucciones en el conjunto $(P, Q, R, \ldots)$*. El resto de instrucciones del programa pueden ejecutar simultáneamente sin ningún problema.

Se llama **región crítica** a esa región del programa donde se debe proporcionar exclusión mutua para evitar una colisión en la utilización de un recurso compartido. Ten en cuenta que la **región crítica** no tiene por qué ser una sola secuencia de instrucciones del programa, pueden ser varias y sólo debería haber un flujo de código en total ejecutando cualquiera de ellas.

Para proporcionar esta propiedad, se comprueba que no hay ningún flujo accediendo concurrentemente a la región crítica, y para ello necesitamos una operación que **compruebe y escriba la variable `busy` de forma atómica**[8].

Una posible solución podría ser inhabilitar interrupciones para que el flujo no sea expulsado de la CPU mientras esté en esa región, pero eso no funciona en general por dos motivos.

Primero, cada CPU tiene sus propias interrupciones, por lo que no es una solución válida para un ordenador con más de una CPU (un multiprocesador). Prácticamente todas las máquinas actuales tienen más de una CPU o core. El problema es que A y B pueden estar ejecutando en paralelo, esto es, cada uno en una CPU.

Segundo, no es deseable que los procesos de usuario puedan desactivar todas las inte-

[6] Observa que hablamos en términos de instrucciones ejecutadas, no de flujos de ejecución. Con dos flujos A y B, uno ejecutando la región P y el otro la región Q, la secuencia $[p_1, q_1, p_2 \ldots]$ sería en realidad $[p_{1,A}, q_{1,B}, p_{2,A} \ldots]$

[7] Para dos flujos de ejecución, A y B sería $[p_{1,A}, p_{1,B}, p_{2,A} \ldots]$.

[8] En realidad es más sutil que esto, recuerda el Ejemplo 3. Tiene que sincronizarse todo lo que hay pendiente en la memoria y el compilador tiene que estar al tanto.

rrupciones de la CPU, porque en caso de un error de programación pueden dejar todo el sistema bloqueado. Por ejemplo, si un programa de usuario inhabilita interrupciones y se mete en un bucle infinito, el kernel nunca puede tomar el control para planificar otros procesos o matar al proceso erróneo.

Por tanto, necesitamos soporte del hardware para esto. Todas las CPUs modernas tienen instrucciones que permiten hacer esto[9] (directamente o con instrucciones equivalentes). Esas instrucciones se encargan de cerrar el bus de memoria, sincronizar las cachés, evitar interrupciones, etc. para asegurar la atomicidad. Hay que recordar que no debería haber parte de los efectos laterales (estados intermedios) de un flujo que no se hayan completado cuando entra otro en la región crítica[10]

La operación de la que estamos hablando se llama ***T**est-**a**nd-**S**et* (**TAS**). La operación sólo tiene un operando, un valor booleano que se pasa por referencia. La semántica de esta operación es sencilla:

- Si está a *false*, lo pone a *true* y retorna *false*.
- Si está a *true*, lo pone a *true* y retorna *true*.

Se puede ver así: un flujo asigna *true* a la variable booleana y mediante el valor de retorno comprueba si ha cambiado el valor él mismo (si ya estaba a *true*, ha sido otro flujo).

Por ejemplo, los procesadores Motorola 68000 ofrecen una instrucción TAS. Otros procesadores incluyen instrucciones equivalentes con las que se puede implementar TAS. Por ejemplo, la aquitectura AMD64 tiene una instrucción llamada `XCHGL` (*exchange long*) que intercambia de forma atómica dos valores. Esa instrucción utiliza los mecanismos de cierre del procesador para garantizar que se intercambian los dos valores y garantiza que la memoria está sincronizada entre los diferente procesadores al acabar de ejecutar. Con ella, se puede implementar TAS en ensamblador. Esta es la implementación de TAS en el sistema operativo Plan 9 para la arquitectura AMD64[11]:

[9] Para ver más detalles de cómo se implementa TAS en diferentes juegos de instruccciones de diferentes arquitecturas, ver [28, Capítulo 5.5]

[10] Ni siquiera todo esto es suficiente. El compilador debe estar al tanto de todo esto. Es normal que un compilador optimizador reordene instrucciones para hacer el código más eficiente siempre que no haya una dependencia de datos. El compilador debe saber que estas instrucciones no se pueden reordenar o proporcionar barreras o invariantes (en las llamadas a función, por ejemplo) que permitan evitar que los efectos de las instrucciones de la región crítica se mezclen con otras instrucciones o haya reordenaciones peligrosas. Un ejemplo de esto es que el compilador debe generar código que guarde las variables en memoria (aunque estuviesen en registros) para que puedan tener efecto las sincronizaciones de las instrucciones especiales.

[11] En el compilador de Plan 9 se garantiza que las llamadas a función son barreras infranqueables para la reordenación de instrucciones. Si una variable va en un registro, se guarda en memoria antes de saltar a una llamada a función o retornar y que si se extrae la dirección de una variable, ésta se encuentra en memoria y no en un registro. Esto no es así en todos los compiladores ni en todos los lenguajes. Cada uno tiene invariantes, garantías y funciones especiales para estas tareas.

```
1 TEXT      _tas(SB),$0
2           MOVL        $0xdeadead,AX
3           MOVL        1+0(FP),BX
4           XCHGL       AX,(BX)
5           RET
```

Ese código escribe la constante `0xdeadead` en el registro AX, escribe el puntero que se le ha pasado como argumento en el registro BX, intercambia el valor de AX con el valor apuntado por el puntero en BX y retorna el contenido actual de AX.

Si el valor apuntado tenía un 0, se retorna 0 (*false*) y ahora tiene 0xdeadead (*true*). Si el valor apuntado tenía un valor distinto de 0 (*true*), lo devuelve y ahora tiene 0xdeadead (*true*).

Con una operación TAS ya podemos tener exclusión mutua, pero es deseable tener una operación de más alto nivel que sea más fácil de usar en nuestros programas. Con TAS podemos implementar **locks** (cierres o cerrojos).

7.3. Locks (cerrojos, cierres)

Un *lock* tiene dos operaciones:

- `lock`: coger[12] el cierre. Si un cierre está libre, se puede coger. Si se intenta adquirir un cierre ya cogido, el flujo se queda bloqueado hasta que pueda cogerlo.
- `unlock`: suelta el cierre. Nunca se debe intentar soltar un cierre que no se ha adquirido previamente.

Para entrar en la región crítica (esto es, para acceder al recurso compartido), hay que tener cogido el cierre. Después de utilizar el recurso compartido[13], siempre se debe soltar el cierre.

Otra manera de verlo es que entre la ejecución de `lock` y `unlock` hay un fragmento de la región crítica. Todos los fragmentos van a tener `lock` antes y `unlock` después sobre el mismo cerrojo. El cerrojo identifica la región crítica. Al entrar en dicha región, va a haber una rotura de invariantes, estados intermedios compartidos que no deberían existir cuando se salga de la región crítica. Estos estados intermedios no deben ser visibles para ningún otro flujo de ejecución que haya adquirido el mismo cierre mediante `lock`. Al acabar la región crítica, dichas invariantes se cumplirán. Un ejemplo de invariante es: "la variable compartida contiene un valor válido". Este invariante sólo es importante

[12] Utilizaremos de forma indistinta para para expresar la acción de coger el cierre las palabras adquirir, coger, echar y cerrar y sus antónimos para soltar el cierre. Unas vienen de la metáfora del cerrojo (echar, cerrar), otras de la idea de un sistema de relevos en el que sólo puede actuar el que tiene un objeto físico (adquirir, coger). Esto es normal en la literatura.

[13] Cuando el recurso compartido es una variable o un trozo de memoria, también recibe el nombre de *estado mutable compartido*. Mutable significa que se puede cambiar, asignándole nuevos valores. Cuando la memoria es de sólo lectura (y ningún hilo puede modificarla nunca), no es necesario protegera. Ojo, esto no es así si un hilo accede para leer y otro para escribir.

para las regiones de código que van a acceder a dicha variable y que estarán protegidas por el mismo cerrojo.

Hay que tener en cuenta que la relación entre las variables (o el estado compartido) y el cerrojo está únicamente en la cabeza del programador y es importante dejarlo claro mediante el nombre de las variables, estructura de datos o comentarios (o las tres cosas).

En el mismo escenario que el Ejemplo 1, con una variable compartida adicional `lk` de tipo cerrojo:

```
1 lock(&lk);
2 x++;
3 printf("%d", x);
4 unlock(&lk);
```

¿Se ha eliminado la condición de carrera? Sí. Mientras que un flujo está en la región crítica (el código entre la llamada a `lock` y la llamada a `unlock`), el otro flujo no puede entrar (se quedará bloqueado en la llamada a `lock`)[14].

Ahora el contador es coherente. Ya sólo pueden pasar dos cosas:

- A imprime 1 y B imprime 2.
- A imprime 2 y B imprime 1.

Ya no podemos ver valores incorrectos o perder incrementos. Ten en cuenta que ese código no impone un orden específico (p. ej. que A sea el primero en incrementar).

Cuando necesitamos proteger distintos recursos compartidos se pueden tener distintas regiones críticas con distintos cerrojos, para optimizar el uso concurrente de los recursos que son independientes. Cuando usamos distintos cerrojos, aparece el problema de los interbloqueos (deadlocks). Supongamos esta secuencia:

1. A coge lk1, espera a que B suelte lk2.
2. B coge lk2, espera a que A suelte lk1.
3. A espera a B, pero B no acaba porque espera a A. Esto es una dependencia circular. **Ninguno progresa**.

Por ejemplo, esto puede pasar cuando se necesita coger dos cerrojos y se hace en distinto orden en dos partes distintas del código como se muestra en la Figura 7.1.

En general, las buenas prácticas a la hora de usar cerrojos son:

- Coger los distintos cierres siempre en el mismo orden.
- Tener el `lock()` y el `unlock()` en la misma función.
- Intentar tener el `lock()` y el `unlock()` al mismo nivel de tabulación.

[14] Observa que la relación entre `x` y `lk` está en la cabeza del programador. No estaría mal haber llamado a la variable `xlk`, o meter ambas en una struct o poner un comentario o las tres cosas.

```
// A ejecuta...

lock(&lk1);
lock(&lk2);
//acceso a los recursos
//por ejemplo x1=x2
unlock(&lk2);
unlock(&lk1);
```

```
// B ejecuta...

lock(&lk2);
lock(&lk1);
//acceso a los recursos
//por ejemplo x1=x2
unlock(&lk1);
unlock(&lk2);
```

Figura 7.1: El orden en el que se adquieren los cierres sí importa.

- Todos los cierres se usan al mismo nivel de abstracción. Si tenemos funciones de distintos niveles de abstracción, se deben utilizar los cierres al mismo nivel.
- Tener bien claro qué funciones cogen el lock, cuáles se tienen que llamar con el lock cogido, etc.

Por ejemplo, supongamos el siguiente código:

```
1  lock(&lk);
2  if (bla){
3          dosomething();
4          doanotherthing();
5  }else{
6          return;
7  }
8  unlock(&lk);
```

Esto está mal, es incorrecto. Si se entra por la rama del *else*, retornamos de la función sin soltar el cerrojo. Esto va a provocar que todos los flujos que intenten coger el cerrojo se queden bloqueados de por vida.

Si lo cambiamos:

```
1  lock(&lk);
2  if (bla) {
3          dosomething();
4          doanotherthing();
5  }
6  else {
7          unlock(&lk);
8          return;
9  }
10 unlock(&lk);
```

Ahora está mejor, pero sigue estando mal (es correcto, pero hace fácil cometer un error si se modifica el código más adelante).

Este código es mejorable. Tenemos llamadas a `lock` y `unlock` a distinto nivel de tabulación y se puede simplificar.

```
1 lock(&lk);
2 if (bla) {
3         dosomething();
4         doanotherthing();
5 }
6 unlock(&lk);
7 return;
```

Ahora está mucho mejor. Tenemos que tener en cuenta que no siempre podemos hacer esto y aplicar todas las buenas prácticas adecuadas dadas las características del código.

7.3.1. Accesos de lectura

Una observación importante, es que, para acceder a una variable compartida de la que solo se va a leer (y que otros hilos van a actualizar), también es necesario adquirir un `lock`.

Por ejemplo supongamos que tenemos dos flujos de control y una variable compartida `x`. Uno flujo ejecuta:

```
1 lock(&xlk);
2 x = y;
3 unlock(&xlk);
```

Y el otro:

```
1 lock(&xlk);
2 printf("%d", x);
3 unlock(&xlk);
```

Parece que no es necesario adquirir el cerrojo `xlk` en el segundo trozo de código, puesto que sólo se accede a su valor. Sin embargo, no hacerlo es peligroso. Lo primero es que, dependiendo de la arquitectura, la variable se puede actualizar en varias operaciones no atómicas (por ejemplo, primero la parte baja y luego la alta ; depende de la anchura del bus de datos y del tamaño de la variable). Además, puede suceder que se esté accediendo a una versión de la caché local de este procesador y sea un valor antiguo. En general, es probable que termine imprimiendo valores erróneos o antiguos. Incluso si estudio bien la arquitectura y parece que me ofrece las garantías adecuadas, sigue siendo peligroso. Si lo dejo sin cerrojo, código no es portable y es muy frágil (cambiar un `int` por un `long long int` puede cambiar el comportamiento, por ejemplo).

7.4. Tipos de locks

La **contienda** (*contention*) es la cantidad de competencia entre los flujos concurrentes por adquirir un recurso (o un cierre). Cuando hay muchos que quieren entrar a la región crítica, sólo uno puede entrar y el resto se tendrán que esperar. Cuando muchos flujos tienen que esperar, la contienda es elevada. Eso es malo en general, se va a tardar más en realizar las tareas, ya que habrá flujos esperando.

Por eso, siempre tenemos que intentar que las regiones críticas sean lo más pequeñas posible: dentro sólo se debe hacer lo indispensable para que los flujos salgan cuanto antes y el resto no tenga que esperar tanto. Al final, como en todos los problemas de este tipo, habrá compromisos entre latencia (el tiempo que se espera de una forma o de otra) y *throughput* (el trabajo total realizado). Estos compromisos dependerán del nivel de contienda. Un ejemplo de esto sucede cuando una región crítica se parte en dos más pequeñas (suponiendo que sea posible hacer esto correctamente). Se repartirá mejor entre los flujos el tiempo de espera. En particular, seguramente el tiempo de espera máximo disminuirá. A cambio, se pagará el coste de las operaciones de `lock` y un `unlock` y el *throughput* total del sistema puede disminuir[15].

En general, hay dos tipos de cerrojos: **spin-locks** y **queue-locks**.

7.4.1. Spin locks

En un **spin-lock**, el flujo que se bloquea en el cerrojo hace **espera activa**, también llamada *polling*: el flujo consume tiempo de CPU iterando hasta poder coger el cerrojo. Dicho de otra forma, está iterando e iterando intentando adquirir el cerrojo. Por tanto, pueden llegar a consumir mucha CPU.

Sólo se deben usar cuando hay **poca contienda** y la región crítica es pequeña. En este caso son útiles porque no necesitan la intervención del kernel (por tanto sus operaciones tienen un coste menor en el caso de que el hilo no tenga que esperar). Si se usan cuando hay mucha contienda, los hilos estarán consumiendo mucho tiempo de CPU intentando entrar en la región crítica. Al hacer *polling*, hay un compromiso entre latencia y uso de la CPU durante la espera.

Aparte de lo anterior, no son justos. Esto quiere decir que hay flujos que se pueden ver perjudicados y tardar mucho más que el resto o no coger nunca el cierre (*hambruna* o *starvation*).

A continuación se muestra la implementación de *spin-locks* del sistema operativo Plan 9, que usa la implementación de TAS que hemos visto antes:

Programa de C 7.1: *spin-locks* en Plan 9

```
1 void
2 lock(Lock *lk)
3 {
```

[15] O no, dependiendo de si se usan estrategias de *backoff* como veremos con los **spin-locks** y cómo de importante sea su efecto dependiendo de la contienda.

```
4           int i;
5
6
7           if(!_tas(&lk->val))             /* once fast */
8                   return;
9           for(i=0; i<1000; i++){          /* a thousand times pretty fast */
10                  if(!_tas(&lk->val))
11                          return;
12                  sleep(0);
13          }
14          for(i=0; i<1000; i++){          /* now nice and slow */
15                  if(!_tas(&lk->val))
16                          return;
17                  sleep(100);
18          }
19          while(_tas(&lk->val))           /* take your time */
20                  sleep(1000);
21  }
22
23  void
24  unlock(Lock *lk)
25  {
26          /* on AMD64, this is atomic */
27          lk->val = 0;
28  }
```

La estructura `Lock` sólo tiene un campo, su valor (booleano que indica si el cerrojo está cogido o no). Esta implementación de *spin-locks* introduce un *backoff*: al principio empieza con un número grande de intentos por unidad de tiempo y va insistiendo cada vez menos a menudo. Prueba una vez, después prueba 1000 veces cediendo el resto del cuanto del procesador en cada iteración (llamar a `sleep` con 0 hace que se planifique el proceso actual), después prueba 1000 veces esperando 100 milisegundos[16] y, finalmente, espera 1 segundo antes de cada intento para entrar iterando hasta conseguirlo. La función `unlock()` simplemente asigna *false* al valor del cierre, teniendo en cuenta que en AMD64 la asignación de un `int` es atómica (otras arquitecturas tendrán, si es necesario, otras implementaciones de esta función)[17].

7.4.2. Queue locks (mutex)

En los **queue-locks** (también conocidos como **mutex**[18]), el flujo que se bloquea en el cerrojo suelta el procesador y pasa a un estado que no es *listo para ejecutar*. El planificador no lo seleccionará para ejecutar hasta que pueda adquirir el cerrojo. Por tanto, estos cerrojos no hacen *espera activa*. Cuando se pueda coger el cierre, el flujo de control volverá a estar *listo para ejecutar* y el planificador le asignará una CPU cuando llegue su turno.

[16] En Plan 9 la llamada al sistema `sleep` admite el tiempo en milisegundos, no en segundos como en Unix.

[17] Puede hacer esto gracias a las garantías sobre las variables cuyo valor está temporalmente en registros y las reordenaciones de instrucciones del compilador de Plan 9: Garantiza que no cruzan las fronteras de una llamada a función.

[18] *Mutex* significa en general ***Mut****ual* ***Ex****clusion*, es decir, exclusión mutua.

Como estos cerrojos tienen que dejar bloqueados los flujos, necesitan la intervención del kernel (esto es, hay el proceso tiene entrar al kernel para poder pasar su estado de planificación a *bloqueado*). Por eso puede dar la impresión de que son *"menos rápidos"* que los *spin-locks*. Hay que tener cuidado con esa afirmación: depende del programa, del tamaño de la región crítica y del nivel de contienda: establecen compromisos diferentes. Más adelante veremos un ejemplo.

Estos cerrojos son justos, y suelen ser FIFO (aunque esto depende de la implementación), esto es, el primero que se bloquea en el cerrojo es el primero que lo adquiere cuando lo suelte el que lo tiene ahora. Nótese que pueden ser justos (en el sentido de que no hay hambruna) aunque no implementen una cola estricta (FIFO).

7.4.3. Cerrojo de lectores/escritores

Otro tipo de primitiva es el **cerrojo de lectores/escritores**. En este caso, hay dos tipos de operaciones para adquirir el cerrojo: para leer el recurso protegido (lectores) y para escribirlo (escritores).

- Los que van a leer el recurso compartido pueden hacerlo concurrentemente (N lectores), ya que no van a modificarlo (y, por tanto, no pueden causar ningún error a los otros lectores). Los lectores cogen el cierre en modo lectura.
- El que va a escribir necesita estar sólo en la región crítica. No puede haber nadie, ni leyendo ni escribiendo el recurso. Coge el cierre en modo escritura.

Observa que, en ambos casos, el proceso llama a las funciones `lock` y `unlock` (tanto para escritores como para lectores). De esta manera, la memoria estará actualizada y no habrá problemas como los descritos en la sección 7.3.1.

Los escritores tienen exclusión mutua con todos los lectores y entre sí. Sin embargo, los lectores no tienen exclusión mutua entre sí. Esta primitiva hace que el nivel de concurrencia sea mayor, es decir, que no haya muchos procesos esperando cuando hay muchas lecturas y pocas escrituras (que es un caso común).

7.4.4. Otras primitivas

Aunque en este capítulo de introducción a la programación concurrente sólo vamos a tratar con cerrojos, hay muchas otras primitivas de sincronización diferentes:

- Futex
- WaitGroup
- Semáforos
- Barreras
- Rendezvous

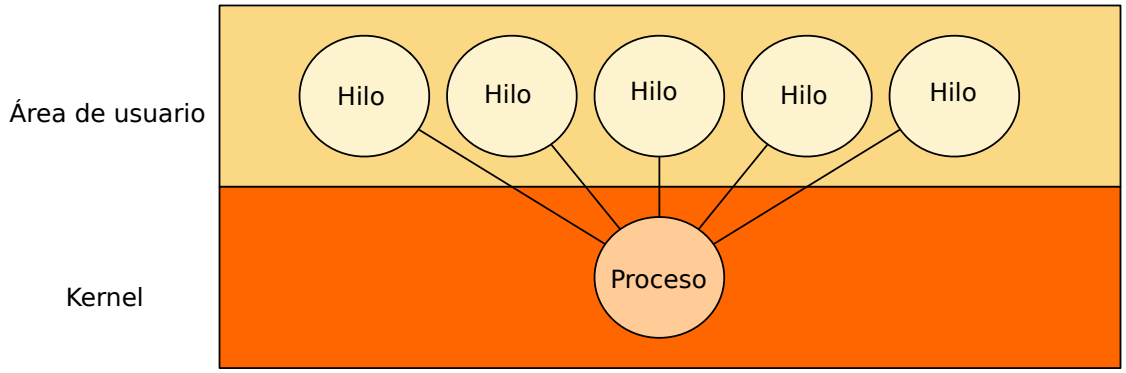

Figura 7.2: Hilos de biblioteca (N-1).

- Variables condición y monitores
- Canales
- ...

Para desarrollar programas concurrentes, es importante estudiar detenidamente los mecanismos ofrece el sistema en el que trabajamos y programar con **mucho cuidado**.

7.5. Threads

Veamos ahora se puede implementar programas con múltiples flujos que compartan memoria. En general, un hilo (*thread*) es un flujo de control que comparte memoria (y muchas otras cosas) con otros hilos. Su contexto es básicamente el estado de la CPU.

Hay distintas formas de implementar hilos:

- **Threads de usuario (o de biblioteca), modelo N-1**. Están completamentente implementados en área de usuario. Es lo que se denomina modelo N-1, múltiples hilos se reparten el flujo de control de un proceso del sistema operativo.

  En kernel no sabe nada sobre los hilos, es algo que se implementa en espacio de usuario. La Figura 7.2 muestra este esquema.

  Es barato crear, destruir y conmutar entre hilos en este modelo. Sin embargo, los hilos correspondientes a un mismo proceso no pueden ejecutar en paralelo cuando disponemos de múltiples procesadores, ya que el sistema operativo no está al tanto de su existencia y no los puede planificar por separado. La biblioteca planifica sus hilos multiplexando el flujo del proceso, cambiando su contexto (que es básicamente el contador de programa y el puntero de pila) con funciones clásicas de Unix como `setjmp` y `longjmp` o las funciones más modernas de la familia `ucontext` (ver *getcontext(3)* y *makecontext(3)*).

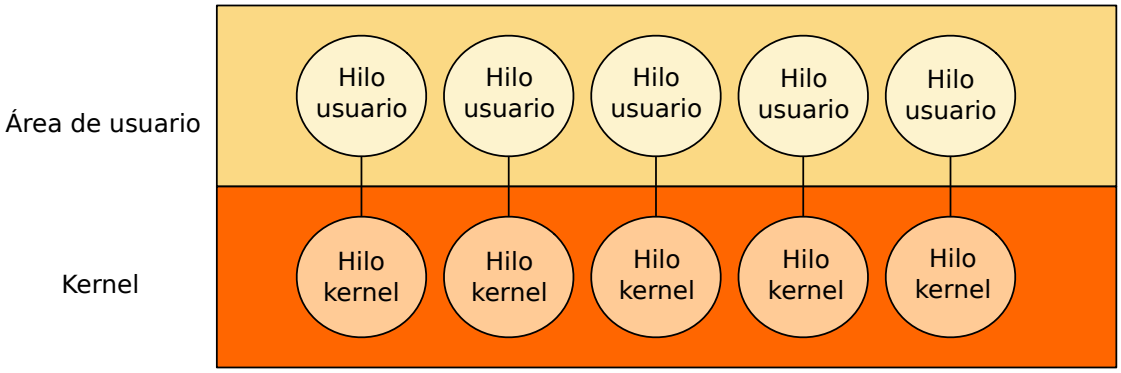

Figura 7.3: Hilos de kernel (1-1).

En este tipo de biblioteca, si se bloquea un hilo (p. ej. haciendo entrada/salida), todos los hilos se bloquearán también (porque en realidad se bloquea el proceso del sistema).

- **Threads de kernel, modelo 1-1**. El sistema operativo implementa los hilos. En este caso, un hilo es un *proceso* que comparte memoria (y otras cosas) con otros procesos.

  La terminología puede ser diferente en distintos sistemas: dependiendo del sistema en el que estemos, a ese flujo se le puede denominar proceso, hilo, etc. La cuestión es que, en este modelo, un hilo es una entidad de planificación del sistema operativo: el planificador del sistema operativo tiene en cuenta esos flujos y los planifica (como vimos con los procesos). La Figura 7.3 muestra este esquema.

  Crear, destruir y planificar los hilos en este modelo es más caro, ya que hay que entrar al kernel. Por otro lado, los hilos pueden bloquearse sin problema y pueden ejecutar en paralelo en distintas CPUs.

- **Modelo híbrido, modelo N-M**. Mezcla las dos aproximaciones anteriores. Un *proceso* puede albergar uno o varios hilos.

  Han existido distintas implementaciones de esta idea, por ejemplo la biblioteca de threads de Plan 9, los *procesos ligeros* (***L**igth**w**eight **P**rocess* o **LWP**) de SunOS o las *gorutinas* del lenguaje de programación Go de Google. La Figura 7.4 muestra este esquema.

  Por ejemplo, en los *LWP*[19] del sistema operativo SunOS, un *proceso* se compone de un espacio de direcciones y un conjunto de LWP que el kernel planificará por separado. Un thread está totalmente representado en espacio de usuario. Un LWP es como una *CPU virtual* para un thread. Los LWP pueden planificar los threads sin necesidad de entrar al kernel. Si un LWP se bloquea, otro LWP puede ejecutar

[19] En general, LWP significa Light-Weight Process, proceso ligero. Tiene distintas connotaciones dependiendo del sistema.

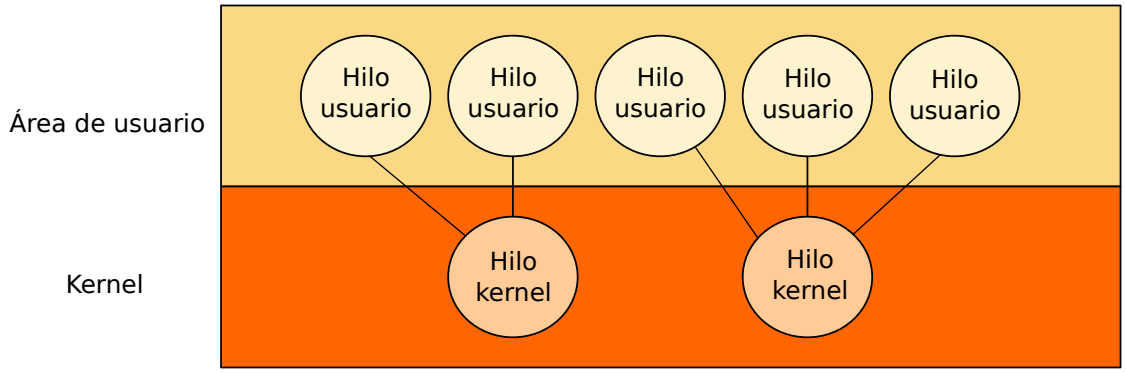

Figura 7.4: Hilos N-M.

el resto de threads del proceso. Cada LWP puede hacer llamadas al sistema, tener fallos de página y ejecutar en paralelo independientemente.

En ocasiones, los threads de usuario reciben el nombre de *green threads* y los de kernel *red threads*. Otra característica de la implementación de threads de unos tipos u otros es si la planificación de los mismos (ya sea el planificador de espacio de usuario o de kernel) es colaborativa o expulsiva, conceptos que vimos en el capítulo 3. Estas dos clasificaciones (*green threads* vs. *red threads* y expulsivos vs. colaborativos) no encajan bien con la clasificación anterior, son propiedades de las diferentes implementaciones (pero tienen consecuencias prácticas para el usuario). Por ejemplo, en una librería colaborativa, un bucle infinito puede hacer que no se puede planificar un thread[20].

En Linux existe una implementación nativa del modelo 1-1 desde hace tiempo. Para ello existe una llamada al sistema, `clone`, que es similar a `fork` pero permite crear un *proceso* hijo que comparte recursos con el padre:

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ... );
```

Tiene muchas *flags* (ver *clone(2)*) para compartir distintos recursos en padre e hijo. Por ejemplo, `CLONE_VM` hace que padre e hijo compartan memoria. Si usamos esa flag, se comparte todo el espacio de direcciones: los mmap/munmap afectan a los dos, las pilas se sitúan en distintas direcciones (el segmento de pila se comparte, con lo que si las pilas no estuviesen en direcciones separadas, sería un problema, la llamadas a función de un hilo pisarían con sus registros de activación la pila del otro hilo), etc. Usar `clone` directamente puede ser complejo. Es más sencillo usar bibliotecas que se apoyan en esta llamada al sistema [32, Capítulo 28]. En Linux, podemos crear programas en C con hilos usando la biblioteca Linux 2.6 NPTL (***N**ative **P**osix **T**hread **L**ibrary*).

En Linux, cada hilo tiene un identificador llamado *Thread ID* (TID). Todo hilo pertenece a un proceso. Un proceso al menos tiene un hilo. Un proceso monohilo tiene un thread con su TID igual al PID.

[20] Por ejemplo, las gorutinas, que son N-M, comenzaron siendo colaborativas. Más tarde se hizo que fuesen expulsivas basándose en una señal periódica (como las que vimos en la sección 6.4.2).

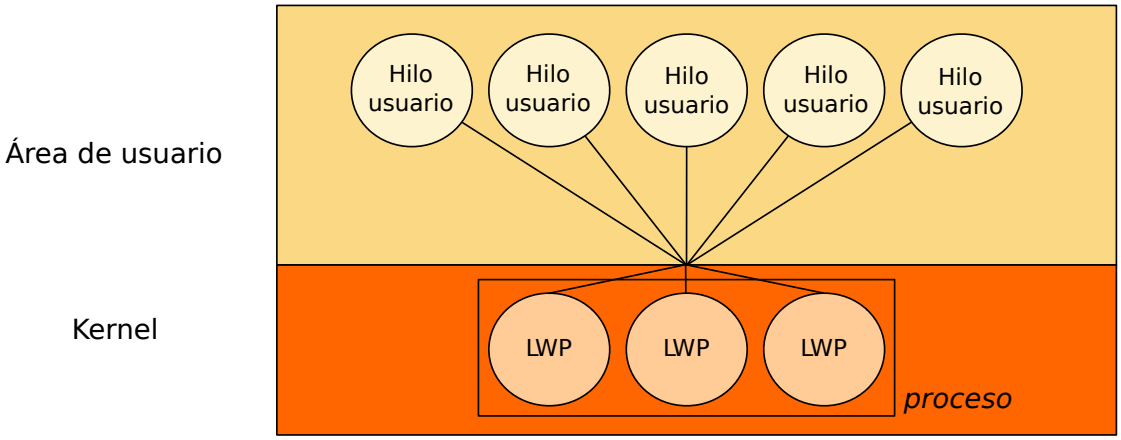

Figura 7.5: Ejemplo de modelo N-M (LWP de SunOS).

Con el comando `ps` podemos ver los hilos asociados a cada proceso. En este ejemplo, el proceso con PID 31009 está ejecutando la aplicación `firefox` y tiene múltiples hilos. En la columna LWP se muestra el TID[21]. El proceso con PID 2486 está ejecutando un `cat` (un programa monohilo), por tiene un único hilo con el mismo TID.

```
 $ ps -q 31009  -eLfm
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
esoriano 31009  5070     -  0   33 18:04 ?        00:00:04 /usr/lib/firefox/firefox
esoriano     -     - 31009  0    - 18:04 -        00:00:03 -
esoriano     -     - 31012  0    - 18:04 -        00:00:00 -
esoriano     -     - 31014  0    - 18:04 -        00:00:00 -
esoriano     -     - 31015  0    - 18:04 -        00:00:00 -
esoriano     -     - 31016  0    - 18:04 -        00:00:00 -
esoriano     -     - 31017  0    - 18:04 -        00:00:00 -
esoriano     -     - 31018  0    - 18:04 -        00:00:00 -
esoriano     -     - 31019  0    - 18:04 -        00:00:00 -
esoriano     -     - 31020  0    - 18:04 -        00:00:00 -
esoriano     -     - 31021  0    - 18:04 -        00:00:00 -
 $ ps -q 2486  -eLfm
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
esoriano  2486 31557     -  0    1 18:25 pts/2    00:00:00 cat
esoriano     -     -  2486  0    - 18:25 -        00:00:00 -
```

7.5.1. Biblioteca pthread en Linux: NPTL

El estándar POSIX Threads es una interfaz de programación para usar hilos, no impone una implementación en concreto. Cada sistema puede implementarla de una forma distinta. En Linux, la página de manual *pthreads(7)* nos da información sobre su implementación. Actualmente, se usa la implementación Linux NPTL[22].

[21] Puede referirse al Thread ID como TID, LWP o SPID (ver *ps(1)*).
[22] La implementación antigua se llamaba `LinuxThreads`, ya no se usa.

Un hilo forma parte de un *thread group*, todos los hilos de ese grupo pertenecen al mismo *proceso* (todos tienen el mismo PID, y cada uno tiene su TID). Cada hilo tiene su propia copia de:

- Variable `errno`.
- Máscara de señales.
- Pila de señales (si se usa una separada).

Internamente, los hilos se crean con la llamada `clone` y como hemos visto anteriormente, los hilos comparten TEXT, DATA, BSS, etc. Cada uno tiene su pila, y todos pueden acceder a las pilas de los demás aunque no es recomendable. Por ejemplo, aunque *"funcione"*, nunca se debería pasar a otro hilo un puntero a una variable local... ¿qué sucede si retorna de la función y después el otro hilo accede a esa variable local (que ya no tiene sentido)? Nunca se deben compartir punteros a variables locales entre distintos hilos.

Como ya se ha dicho anteriormente, cada vez que se acceda a un recurso compartido (p. ej. una variable compartida) hay que protegerse mediante un mecanismo de sincronización (en este tema sólo hemos visto los cerrojos). En caso contrario, surgen problemas. A continuación veremos cómo se utilizan los cerrojos en la biblioteca `pthread`.

Otra cosa que hay que tener en cuenta es que, si se llama a funciones de biblioteca, siempre hay que estar seguro de que dichas funciones son *thread_safe*. Eso significa que la función está preparada para llamarse concurrentemente. Con ese fin, hay que comprobar en su página de manual si es segura en entornos multi-hilo o tiene una versión segura. En caso de duda, siempre hay que protegerse.

Para enlazar un programa con la librería `pthread`, es necesario especificarlo explícitamente:

```
$ gcc -c -Wall -Wshadow -g miprograma.c
$ gcc -o miprograma miprograma.o  -lpthread
```

La interfaz de la biblioteca es bastante grande. Veremos únicamente las funciones básicas:

- `pthread_create`: crea un thread, que comenzará ejecutando la función principal que se le pasa como su tercer parámetro. El primer parámetro es un puntero a la variable de tipo `pthread_t` que identificará al thread creado.

  Un hilo termina llamando a la función `pthread_exit` o retornando de su función principal.

  Si cualquiera de los hilos llama a `exit`, se acabará la ejecución de todos los hilos. Lo mismo pasa si el hilo principal retorna de la función `main`.

- `pthread_join`: espera a que muera el thread indicado en su primer parámetro[23].

Las funciones de `pthread` no ponen `errno` (ver *pthreads(7)*). Veamos un ejemplo:

Programa 7.2: pthreads.c

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <errno.h>
5  #include <string.h>
6  #include <err.h>
7
8  void *
9  fn(void *p)
10 {
11     fprintf(stderr, "Hi! I'm fn\n");
12     return NULL;
13 }
14
15 int
16 main(int argc, char *argv[])
17 {
18     pthread_t thread;
19
20     if (pthread_create(&thread, NULL, fn, NULL) != 0) {
21         warnx("error creating thread");
22         return 1;
23     }
24     fprintf(stderr, "Hi! I'm main\n");
25     if (pthread_join(thread, NULL) != 0) {
26         warnx("error joining thread");
27         return 1;
28     }
29     return 0;
30 }
```

Este programa tiene un hilo principal (el que empieza a ejecutar el programa) y crea un nuevo hilo. El nuevo hilo empezará ejecutando la función `fn`, a la que se le ha pasado un valor `NULL` como argumento para el primer parámetro. Ambos hilos escribirán por su salida estándar un mensaje (cada uno con el nombre de la función que está ejecutando). Luego, el hilo principal espera a que el hilo creado termine. Cuando esto pasa, el hilo principal también termina. Veamos su ejecución:

```
$ gcc -Wall -Wshadow pthreads.c
$ gcc -o pthreads -lpthread
$ ./pthreads
Hi! I'm main
Hi! I'm fn
$
```

[23] Esta llamada nos proporciona una nueva garantía diferente a las vistas anteriormente (atomicidad, exclusión mutua). Garantiza que todas las instrucciones del hilo al que se espera (y sus efectos) se han completado antes de volver de la llamada.

Veamos otro programa que use una variable compartida, `counter`. Se van a lanzar 200 hilos que incrementarán la variable, cada uno un total de 10000 veces. Por tanto, al final el contador tendría que valer 2000000:

Programa 7.3: pthreads-race.c

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <errno.h>
5  #include <string.h>
6  #include <err.h>
7
8  enum {
9      Nthreads = 200,
10     Nrounds = 10000,
11 };
12
13 int counter;
14
15 void *
16 fn(void *p)
17 {
18     int i;
19
20     for (i = 0; i < Nrounds; i++) {
21         counter++;
22     }
23     return NULL;
24 }
25
26 int
27 main(int argc, char *argv[])
28 {
29     pthread_t thread[Nthreads];
30     int i;
31
32     for (i = 0; i < Nthreads; i++) {
33         if (pthread_create(&thread[i], NULL, fn, NULL) != 0) {
34             warnx("error creating thread");
35             return 1;
36         }
37     }
38     for (i = 0; i < Nthreads; i++) {
39         if (pthread_join(thread[i], NULL) != 0) {
40             warnx("error joining thread");
41             return 1;
42         }
43     }
44     if (counter != Nthreads * Nrounds) {
45         fprintf(stderr,
46             "MAIN: counter is %d, should be %d\n",
47             counter, Nthreads * Nrounds);
48     } else {
49         fprintf(stderr, "MAIN: counter is %d, OK\n", counter);
50     }
51     return 0;
52 }
```

Si ejecutamos el programa:

```
 $ ./pthreads-race
MAIN: counter is 448320, should be 2000000
 $
```

¿Qué ha pasado? Estamos incrementando `counter` sin protección. Tenemos una condición de carrera (recuerda que el comportamiento puede ser mucho peor, ya que es indefinido).

En la biblioteca tenemos distintos mecanismos de sincronización, entre ellos, cerrojos:

- Spin locks: tipo de datos `pthread_spin_lock_t`.
- Mutex: tipo de datos `pthread_mutex_t`.

Las operaciones básicas del tipo de datos `pthread_spin_lock_t` son:

- `pthread_spin_init`: inicializa el cerrojo. El segundo parámetro indica si se comparte con otros procesos. Para ello hay que usar la constante `PTHREAD_PROCESS_SHARED`. En ese caso, se puede compartir con otros procesos con acceso a esa memoria. Si sólo queremos que esté compartido por los hilos de este proceso, tenemos que usar la constante `PTHREAD_PROCESS_PRIVATE`).

  En general, la inicialización y la destrucción de los mecanismos de sincronización se debe hacer en un entorno en el que *un sólo hilo* utilice dicho mecanismo. Esto se puede hacer antes de crear los hilos o teniéndolos bloqueados de alguna manera mientras se realiza la inicialización. Esto es importante tanto para `pthread_spin_init` como para `pthread_mutex_init`.

- `pthread_spin_lock`: adquiere el cerrojo.
- `pthread_spin_unlock`: suelta el cerrojo.
- `pthread_spin_destroy`: libera los recursos asociados al cerrojo. De nuevo, hay que ser cuidadosos con que ya no haya hilos utilizándolo. Se puede haber esperado a que acaben usando la primitiva `pthread_join`.

Intentemos arreglar la condición de carrera del ejemplo anterior. Hemos declarado una variable de tipo `pthread_spinlock_t` **global** para que sea compartida por todos los hilos. Esto es muy importante, porque si fuese una variable local de cada hilo, cada uno estaría usando su propio cerrojo y no se garantizaría la exclusión mutua. Esa variable se inicializa en el programa principal **antes de crear los hilos**. Esto es importante también. Si se hace después, tendremos una condición de carrera: habrá hilos usando un cerrojo que está sin inicializar.

Programa 7.4: pthreads-spin.c

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

enum {
    Nthreads = 200,
    Nrounds = 10000,
};

int counter;
pthread_spinlock_t lk;

void *
fn(void *p)
{
    int i;

    for (i = 0; i < Nrounds; i++) {
        pthread_spin_lock(&lk);
        counter++;
        pthread_spin_unlock(&lk);
    }
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t thread[Nthreads];
    int i;

    if (pthread_spin_init(&lk, PTHREAD_PROCESS_SHARED) != 0) {
        warnx("can't init lock");
        return 1;
    }
    for (i = 0; i < Nthreads; i++) {
        if (pthread_create(&thread[i], NULL, fn, NULL) != 0) {
            warnx("error creating thread");
            return 1;
        }
    }
    for (i = 0; i < Nthreads; i++) {
        if (pthread_join(thread[i], NULL) != 0) {
            warnx("error joining thread");
            return 1;
        }
    }
    pthread_spin_destroy(&lk);
    if (counter != Nthreads * Nrounds) {
        fprintf(stderr,
            "MAIN: counter is %d, should be %d\n",
            counter, Nthreads * Nrounds);
    } else {
        fprintf(stderr, "MAIN: counter is %d, OK\n", counter);
    }
    return 0;
}
```

Si lo ejecutamos, parece que ya no tenemos el problema.

```
 $ ./pthreads-spin
MAIN: counter is 2000000, OK
 $
```

También podríamos usar un `mutex` para solucionar el problema. Las operaciones básicas de `pthread_mutex_t` son similares a las del *spin lock*:

- `pthread_mutex_init`: inicializa el cerrojo. Si el segundo parámetro se usa para establecer ciertos atributos, si es NULL se ponen los atributos por omisión. No debe haber más de un hilo accediendo al mutex, de forma similar a `pthread_spin_init`.
- `pthread_mutex_lock`: adquiere el cerrojo.
- `pthread_mutex_unlock`: suelta el cerrojo.
- `pthread_mutex_destroy`: libera los recursos asociados al cerrojo. Como ya se ha explicado, no debe haber más de un hilo accediendo al mutex, de forma similar a `pthread_spin_destroy`.

El programa sería:

Programa 7.5: pthreads-mutex.c

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <errno.h>
5  #include <string.h>
6  #include <err.h>
7
8  enum {
9      Nthreads = 200,
10     Nrounds = 10000,
11 };
12
13 int counter;
14 pthread_mutex_t mutex;
15
16 void *
17 fn(void *p)
18 {
19     int i;
20
21     for (i = 0; i < Nrounds; i++) {
22         pthread_mutex_lock(&mutex);
23         counter++;
24         pthread_mutex_unlock(&mutex);
25     }
26     return NULL;
27 }
28
29 int
30 main(int argc, char *argv[])
31 {
32     pthread_t thread[Nthreads];
33     int i;
34
35     if (pthread_mutex_init(&mutex, NULL) != 0) {
36         warnx("can't init mutex");
37         return 1;
38     }
39     for (i = 0; i < Nthreads; i++) {
40         if (pthread_create(&thread[i], NULL, fn, NULL) != 0) {
41             warnx("error creating thread");
42             return 1;
43         }
44     }
45     for (i = 0; i < Nthreads; i++) {
46         if (pthread_join(thread[i], NULL) != 0) {
47             warnx("error joining thread");
48             return 1;
49         }
50     }
51     pthread_mutex_destroy(&mutex);
52     if (counter != Nthreads * Nrounds) {
53         fprintf(stderr,
54             "MAIN: counter is %d, should be %d\n",
55             counter, Nthreads * Nrounds);
56     } else {
57         fprintf(stderr, "MAIN: counter is %d, OK\n", counter);
58     }
59     return 0;
60 }
```

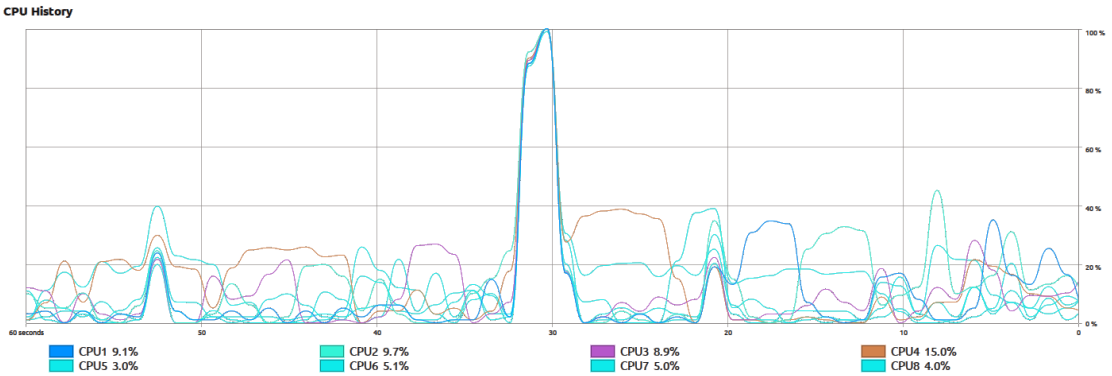

Figura 7.6: El programa con *spin locks* crea un pico de uso de todas las CPUs durante su ejecución; el programa que utiliza *mutex*, que ejecuta después, no genera un pico.

¿Que diferencia hay entre los dos? Ya hemos visto antes los pros y contras de los *spin-locks* y los *queue-locks*. Ejecutemos los dos midiendo el tiempo en terminar:

```
 $ time ./pthreads-spin
MAIN: counter is 2000000, OK

real    0m2.054s
user    0m15.646s
sys     0m0.028s
  $ time ./pthreads-mutex
MAIN: counter is 2000000, OK

real    0m0.197s
user    0m0.290s
sys     0m1.159s
 $
```

Vemos que el programa que usa el *spin lock* ha tardado un orden de magnitud más (tiempo real) que el programa que usa el *mutex*. Además, ha consumido mucho más tiempo[24] de CPU en área de usuario, mientras que el otro ha pasado más tiempo en área de kernel. La figura 7.6 muestra la carga de las CPUs mientras que se han ejecutado los dos programas. El pico que se observa corresponde a la ejecución del programa que usa el *spin-lock*. Aunque la región crítica sea muy pequeña, el programa tiene mucha contienda porque se lanzan muchos hilos concurrentes, que además ejecutan muchas iteraciones.

[24]Ese tiempo es el acumulado por todos los hilos en las distintas CPUs.

7.6. Uso concurrente de ficheros

Ahora que sabemos qué es un cerrojo, ya podemos entender mejor cómo sincronizar los procesos que no comparten memoria pero van a acceder a un fichero concurrentemente y pueden colisionar. Ahora completaremos lo visto sobre ficheros en el capítulo 4.

7.6.1. Creación atómica de ficheros

Un caso típico es la creación de un fichero. ¿Qué pasa si varios procesos quieren crear el mismo fichero concurrentemente?

Esa es la forma de sincronizarse que tienen algunas aplicaciones: si existe un fichero dado, es que ya hay una instancia de la aplicación arrancada. Si no lo hay, se crea el fichero y se arranca la instancia. Esto es lo que hace, por ejemplo, el navegador `firefox` para no arrancar dos navegadores con el mismo perfil. Básicamente, se está usando la existencia de un fichero como un cerrojo entre procesos que no comparten memoria.

Aquí volvemos a tener el mismo problema de antes: necesitamos una operación atómica para comprobar y crear el fichero. Si hacemos una llamada a `access` para comprobar que no existe y en ese caso, hacemos una llamada a `open` para crearlo, tendremos una condición de carrera. Para poder hacerlo atómicamente en una única llamada, tenemos que pasar a `open` las flags `O_CREAT|O_EXCL`. La opción `O_EXCL` hace que si no es esta llamada la que consigue crear el fichero, la llamada a `open` devuelva error y actualice `errno` con el valor `EEXIST`. Es una operación similar a TAS pero para la creación atómica de ficheros. En lugar de comprobar el valor y asignar uno nuevo a una variable booleana de forma atómica, se comprueba la existencia de un fichero y se crea de forma atómica.

7.6.2. Añadir datos de forma atómica

Otro ejemplo de uso concurrente de ficheros es añadir datos al final de un fichero desde múltiples procesos de forma concurrente. Este es el caso de los ficheros de *log*.

Aquí tenemos, otra vez, un problema similar: si queremos añadir datos al final de un fichero, tendríamos que hacer una llamada `lseek` para posicionarnos al final y luego escribir con `write`. Volvemos a tener una condición de carrera: dos procesos se pueden posicionar en el mismo sitio y después escribir allí. Necesitamos situarnos al final del fichero y escribir de forma atómica.

En la llamada al sistema `open` podemos usar la flag `O_APPEND` junto con la de escritura `O_WRONLY`. En ese caso, siempre se escribirá de forma atómica al final del fichero, diga lo que diga el *offset* del descriptor de ficheros.

Nótese que esto sólo asegura que la escritura se realiza al final del fichero. Si un proceso efectúa dos escrituras seguidas, **no** se garantiza que sean contiguas en el fichero (puede colarse la escritura de otro proceso en medio).

7.6.3. Cerrojos para ficheros

Los procesos que no comparten memoria se pueden sincronizar para evitar condiciones de carrera a la hora de escribir y leer ficheros que comparten. La llamada al sistema *flock(2)* sirve para adquirir y soltar un **cierre de lectores/escritores** sobre el fichero[25] que tenemos abierto:

```
int flock(int fd, int operation);
```

Tiene tres operaciones que se especifican en su segundo parámetro con las siguientes constantes: `LOCK_EX` echa el cerrojo como escritor (ni escritores ni lectores van a poder coger el cerrojo mientras que lo tenga), `LOCK_SH` echa el cerrojo como lector (ningún proceso escritor va a poder adquirir el cerrojo mientras, pero otros lectores sí) y `LOCK_UN` suelta el cerrojo que se ha adquirido previamente.

Además, se puede especificar si se desea que la operación sea bloqueante o no. Para especificar que no lo sea, hay que hacer un *or* con la flag `LOCK_NB`. En ese caso, si no se puede coger el cierre, la operación retorna con error. En otro caso, la llamada se bloquea hasta que se pueda adquirir el cerrojo.

Cuando se cierra un fichero que tiene un cerrojo echado, *close(2)* lo suelta al liberar el descriptor de fichero.

Veamos un ejemplo. Supongamos que queremos crear múltiples procesos para que escriban una serie de mensajes (5 en este caso) en un fichero que comparten. Queremos que los 5 mensajes de un mismo proceso estén contiguos en el fichero y que no se pierda ninguno. El fichero puede no existir cuando se ejecuta el programa.

Tenemos el código que se puede ver en la siguiente página:

[25] Hay un único cierre asociado a cada fichero

Programa 7.6: filerace.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/file.h>

enum {
    Nprocs = 10,
    Maxstr = 256,
    Nrounds = 5,
};

void
dowrite(int fd)
{
    char str[Maxstr];
    int i;

    for (i = 0; i < Nrounds; i++) {
        snprintf(str, Maxstr, "PID %d was here! round %d\n",
            getpid(), i);
        if (write(fd, str, strlen(str)) != strlen(str))
            err(EXIT_FAILURE, "can't write file");
    }
}

void
child(char *path)
{
    int fd;

    fd = open(path, O_RDWR | O_CREAT | O_APPEND, 0666);
    if (fd < 0)
        err(EXIT_FAILURE, "can't create file %s", path);
    dowrite(fd);
    close(fd);
}

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

    if (argc != 2)
        errx(EXIT_FAILURE, "usage: %s file", argv[0]);

    unlink(argv[1]);

    for (i = 0; i < Nprocs; i++) {
        switch (fork()) {
        case -1:
            err(EXIT_FAILURE, "can't fork");
        case 0:
            child(argv[1]);
            exit(EXIT_SUCCESS);
        }
    }
    while (wait(NULL) != -1)
        fprintf(stderr, ".");
    fprintf(stderr, "\n");
    exit(EXIT_SUCCESS);
}
```

En este caso, aunque se realicen escrituras *append only*, no se garantiza que las 5 escrituras de cada proceso aparezcan de forma secuencial en el fichero. Si ejecutamos el programa:

```
 $ ./filerace /tmp/f
..........
 $ head -10 /tmp/f
PID 11222 was here! round 0
PID 11223 was here! round 0
PID 11220 was here! round 0
PID 11219 was here! round 0
PID 11221 was here! round 0
PID 11222 was here! round 1
PID 11223 was here! round 1
PID 11224 was here! round 0
PID 11220 was here! round 1
PID 11219 was here! round 1
 $
```

En este caso es necesario usar un cerrojo para garantizar la exclusión mutua mientras que un proceso está escribiendo sus 5 mensajes. Cambiemos la función `child()` de esta forma:

Programa de C 7.7: Función con flock

```
1  void
2  child(char *path)
3  {
4      int fd;
5
6      fd = open(path, O_RDWR|O_CREAT|O_APPEND, 0666);
7      if(fd < 0)
8          err(EXIT_FAILURE, "can't create file %s", path);
9      flock(fd, LOCK_EX);
10     dowrite(fd);
11     flock(fd, LOCK_UN);
12     close(fd);
13 }
```

Si ejecutamos ese programa, podremos comprobar que se han escrito todos los mensajes y son contiguos:

```
 $ ./filerace /tmp/f
..........
 $ head -25 /tmp/f
PID 11325 was here! round 0
PID 11325 was here! round 1
PID 11325 was here! round 2
PID 11325 was here! round 3
PID 11325 was here! round 4
PID 11327 was here! round 0
PID 11327 was here! round 1
PID 11327 was here! round 2
PID 11327 was here! round 3
PID 11327 was here! round 4
PID 11329 was here! round 0
PID 11329 was here! round 1
PID 11329 was here! round 2
PID 11329 was here! round 3
PID 11329 was here! round 4
PID 11333 was here! round 0
PID 11333 was here! round 1
PID 11333 was here! round 2
PID 11333 was here! round 3
PID 11333 was here! round 4
PID 11330 was here! round 0
PID 11330 was here! round 1
PID 11330 was here! round 2
PID 11330 was here! round 3
PID 11330 was here! round 4
 $ wc -l /tmp/f
50 /tmp/f
 $
```

¿Es necesaria la apertura `O_APPEND` en este caso? Sí. Si quitamos la flag `O_APPEND` de la apertura pasaría lo siguiente:

```
 $ ./filerace /tmp/f
..........
 $ cat /tmp/f
PID 11562 was here! round 0
PID 11562 was here! round 1
PID 11562 was here! round 2
PID 11562 was here! round 3
PID 11562 was here! round 4
 $
```

¿Qué ha pasado? Todos los procesos han escrito sus cinco mensajes en el fichero, pero todos lo han hecho desde el *offset* 0. Por tanto, lo que vemos en el fichero al final son los 5 mensajes que ha escrito el último proceso que logró adquirir el cerrojo.

Hay otra forma de adquirir cierres para ficheros. La llamada al sistema `flock` viene de los sistemas BSD (apareció en 4.2BSD). Unix System V tenía otra llamada al sistema *fcntl(2)*. Esta llamada permite tener cierres para distintas regiones de un fichero (no obliga a tener el cerrojo para el fichero completo como hace `flock`). El estándar POSIX.1 eligió esta aproximación. En Linux, y en la mayoría de sistemas actuales, tenemos ambas llamadas.

Hay que tener en cuenta que los cerrojos de `flock` son siempre consultivos (*advisory*). Esto significa que si un proceso decide acceder al fichero sin usar el cerrojo, podrá (no lo debería hacer, pero lo puede hacer). Los cerrojos de `fcntl` pueden ser *advisory* (consultivos) o *mandatory* (obligatorios). En el segundo caso, el kernel comprueba cada llamada a `open`, `read` y `write` para verificar que el proceso no está violando un cerrojo adquirido [19, Capítulo 14]. Algunos sistemas derivados de BSD no implementan esto y sus cerrojos siempre son *advisory*.

Hay que tener cuidado con los cerrojos en determinados sistemas de ficheros. El comportamiento puede cambiar dependiendo del sistema operativo y el tipo (y versión) del sistema de ficheros en el que lo intentamos usar. Por ejemplo, en Linux, hay que tener cuidado en un sistema de ficheros que no sea de la familia EXT. Otro ejemplo: en un sistema de ficheros en red compartido mediante NFS, estas funciones se pueden comportar de una forma inesperada. Por eso es necesario leer con detenimiento la página de manual de *flock(2)* y *fcntl(2)* antes de utilizar estas funciones en NFS.

También hay que ser cuidadoso mezclando el uso de cerrojos sobre ficheros y funciones de la biblioteca *stdio(5)*. Recuerda que esas funciones utilizan el descriptor por debajo para mantener el buffer interno. Por ejemplo, puede suceder que se intente descargar el buffer en el fichero cuando el cerrojo ya no está echado.

El fichero `/proc/locks` nos permite saber qué cerrojos de ficheros se encuentran actualmente adquiridos por algún proceso en un sistema Linux:

```
$ cat /proc/locks
1: POSIX  ADVISORY  WRITE 12893 08:06:4457709 0 EOF
2: POSIX  ADVISORY  WRITE 12893 08:06:4456919 0 EOF
3: POSIX  ADVISORY  WRITE 7236 08:06:11928005 1073741826 1073742335
4: POSIX  ADVISORY  WRITE 7236 08:06:11928004 1073741826 1073742335
5: POSIX  ADVISORY  WRITE 7236 08:06:11927990 1073741826 1073742335
6: POSIX  ADVISORY  WRITE 5743 00:16:841 0 EOF
7: FLOCK  ADVISORY  WRITE 6670 00:35:39 0 EOF
8: POSIX  ADVISORY  READ  6463 08:06:11928702 128 128
9: OFDLCK ADVISORY  WRITE -1 08:06:11927725 0 0
$
```

Después del índice, los campos son: el tipo (`POSIX` y `OFDLCK` son cerrojos de `fcntl`, `FLOCK` de `flock`), si son obligatorios o consultivos, el modo adquirido (leer o escribir), el

PID del proceso que tiene el cerrojo, identificadores del fichero (números *major/minor* del dispositivo e i-nodo del fichero) y los *offset* de comienzo y fin (en `flock`, el fichero completo).

7.7. Ejercicios no resueltos

1. Escribe un programa en C mediante *pthreads(7)* que tenga una condición de carrera sobre un contador compartido (leyéndolo a una variable local y actualizándolo). Consigue que se reproduzca la condición de carrera mediante llamadas a `sleep`. Consigue que se reproduzca la condición de carrera mediante la creación de muchos hilos. Arréglala utilizando *locks*.†∗

2. Escribe un programa en C que se pueda ejecutar varias veces de forma independiente y que tenga una condición de carrera sobre un contador compartido en un fichero (leyéndolo a una variable local y actualizándolo). Consigue que se reproduzca la condición de carrera mediante llamadas a `sleep`. Consigue que se reproduzca la condición de carrera mediante la creación de muchos procesos. Arréglala utilizando *flock*.†∗

3. Busca el fichero de lock de `firefox`. Escribe un programa en C que lo cree al arrancar y lo borre al salir después de un minuto. Comprueba que `firefox` no puede arrancar mientras esté tu programa ejecutando.†∗

4. Escribe en C una versión sencilla de `du`, (que imprime lo que ocupa cada árbol recorriéndolo de forma recursiva, como `du -s`). Esta versión ejecutará un número máximo de pthreads (por ejemplo 16) y usará un thread diferente para cada argumento (si se queda sin threads, cuando un thread acabe se cogerá otro argumento). Esto se llama *pool* de threads. Mide (por ejemplo con `time`) cuanto tarda en ejecutar el programa para diferente número de threads. Escribe una versión que imprima al final la suma de los tamaños calculados por todos los threads. † † ∗

5. Escribe un programa en C que calcule el número Pi mediante una aproximación de montecarlo. Elige dos números en coma flotante entre 0.0 y 1.0 de forma pseudoaleatoria (por ejemplo, con *random(3)*). Calcula qué fracción caen dentro de $x^2 + y^2 < 1$. Mira cuanto tarda el programa en aproximar Pi con un error, por ejemplo $1e-7$. Ejecuta a continuación el programa con varios pthreads generando los números pseudoaleatorios y observa si es más rápido. Prueba con spin-locks, mutex y diferente número de threads.† † ∗

6. Haz que los asignadores que programaste en el capítulo anterior sean *thread-safe* (por ejemplo, usando cierres para las estructuras de datos).†∗

7. Escribe en C una librería de logging que se ocupe de los detalles de concurrencia de manejar un log desde procesos que comparten memoria de forma `thread-safe`. Tendrá tres funciones públicas: `int openlog(char *name);`, `int closelog(int fd);` y `int log(char *format, ...);`. La función `openlog` devolverá el descriptor de fichero en caso de éxito. La función `log` actuará como `printf` para imprimir sobre el log. Todas las funciones devolverán -1 en caso de error. Cuidado con el buffering compartido de entrada salida.† † ∗∗

Bibliografía

[1] A. S. Tanenbaum, *Modern Operating Systems.* USA: Prentice Hall Press, 3rd ed., 2007.

[2] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts Essentials.* Wiley Publishing, 2010.

[3] W. Stallings, *Operating Systems: Internals and Design Principles.* USA: Prentice Hall Press, 6th ed., 2008.

[4] B. Kernighan, *Unix: A History and a Memoir.* Independently Published, 2019.

[5] J. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[6] N. Stephenson, *In the beginning... was the command line.* Avon Books New York, 1999.

[7] B. W. Kernighan and R. Pike, *Unix programming environment.* Prentice-Hall Software Series, 1984.

[8] S. Powers, J. Peek, T. O'Reilly, M. Loukides, and M. K. Loukides, *UNIX power tools.* O'Reilly Media, 2009.

[9] B. W. Kernighan and R. Pike, *The practice of programming.* Addison-Wesley Professional, 1999.

[10] C. Newham and B. Rosenblatt, *Learning the bash shell: Unix shell programming.* O'Reilly Media, 2005.

[11] S. G. Kochan and P. Wood, *Shell Programming in Unix, Linux and OS X: The Fourth Edition of Unix Shell Programming.* Addison-Wesley Professional, 2016.

[12] M. Lam, R. Sethi, J. Ullman, and A. Aho, "Compilers: Principles, techniques, and tools," 2006.

[13] M. Fitzgerald, *Introducing Regular Expressions: Unraveling Regular Expressions, Step-by-Step.* O'Reilly Media, 2012.

[14] J. E. Friedl, *Mastering regular expressions.* O'Reilly Media, 2006.

[15] A. D. Robbins and D. Dougherty, *Sed and Awk.* O'Reilly Media, 1997.

[16] N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME), Part One Format of Internet Message Bodies," RFC 2045, RFC Editor, November 1996.

[17] C. J. Date, *An introduction to database systems*. Pearson, 8th ed., July 2003.

[18] S. Chacon and B. Straub, *Pro git*. Springer Nature, 2014.

[19] W. R. Stevens and S. A. Rago, *Advanced Programming in the UNIX Environment*. Addison-Wesley Professional, 3rd ed., 2013.

[20] D. Brin, *Gente de Barro*. B Nova Ciencia Ficción, 2002.

[21] R. Stallman, R. Pesch, S. Shebs, and M. Free Software Foundation (Cambridge, *Debugging with GDB: The GNU Source-level Debugger*. A GNU manual, GNU Press, 2002.

[22] "Standard RAID levels." Wikipedia. `https://en.wikipedia.org/wiki/Standard/_RAID_levels`.

[23] "Nested RAID levels." Wikipedia. `https://en.wikipedia.org/wiki/Standard/_RAID_levels`.

[24] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. O'Reilly media, 2005.

[25] S. Ghemawat, "tcmalloc." `http://gperftools.github.io/gperftools/tcmalloc.html`.

[26] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?," *ACM Sigplan Notices*, vol. 34, no. 3, pp. 26–36, 1998.

[27] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals." `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`.

[28] J. l. Henessy and D. A. Patterson, *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, 5th ed., 2012.

[29] R. Cox, "Memory models." `https://research.swtch.com/hwmm`.

[30] Go authors, "Data race detector." `https://golang.org/doc/articles/race_detector.html`.

[31] L. Lamport, "TLA$^+$ web page." `http://lamport.azurewebsites.net/tla/tla.html`.

[32] M. Kerrisk, *The Linux Programming Interface*. No Starch Press Series, No Starch Press, 2010.

Agradecimientos

Agradecemos a Alberto Cortés, Felipe Ortega, Guillermo Megias y Alejandro Cruz por las sugerencias y el reporte de errores.

Índice alfabético

Sobre los autores

Enrique Soriano-Salvador es Profesor Titular en la Universidad Rey Juan Carlos de Madrid. Es usuario de sistemas de tipo Unix desde 1998, Ingeniero en Informática desde 2002 y Doctor en Informática desde 2006. Desde 2003, cuando comenzó su Tesis Doctoral con una beca de investigación predoctoral FPI, investiga e imparte docencia sobre sistemas operativos. Ha estado involucrado en el diseño y desarrollo de diferentes sistemas distribuidos, arquitecturas de seguridad y sistemas operativos de investigación (p. ej. Plan B, Nix, Octopus, SHAD, SealFS). Ha impartido cursos de formación de sistemas operativos y seguridad para empresas y administraciones públicas, y desde 2006 imparte docencia universitaria en diversas titulaciones oficiales de Ingeniería Informática e Ingeniería de Telecomunicación, en asignaturas de sistemas operativos, sistemas distribuidos, concurrencia y seguridad. Es autor de múltiples publicaciones científicas en conferencias como *IEEE PerCom*, *Bell Labs Technical Conference* o *International Workshop on Plan 9*, y revistas como *IEEE Pervasive Computing*, *Pervasive and Mobile Computing*, *Journal of Systems and Software*, *Robotics and Autonomous Systems*, *Concurrency Computation Practice and Experience*, *Software Practice and Experience*, *Computers & Security* o *Bell Labs Technical Journal*.

Contacto:
`enrique.soriano@urjc.es`
https://gsyc.urjc.es/~esoriano/
`twitter: @e__soriano`

Gorka Guardiola Múzquiz es Profesor Titular en la Universidad Rey Juan Carlos de Madrid. Es usuario de sistemas de tipo Unix desde 1995, Ingeniero de Telecomunicación desde 2003 y Doctor en Informática desde 2007. Desde 2003, investiga e imparte docencia sobre sistemas operativos. Tuvo el privilegio de realizar una estancia en Bell Labs trabajando en Plan 9, en el laboratorio donde se desarrolló originalmente Unix. Ha estado involucrado en el diseño y desarrollo de software dentro y fuera del kernel para diferentes sistemas distribuidos, sistemas operativos de investigación (p. ej. Plan 9, Plan B, Nix, Octopus, SealFS) y sistemas operativos de uso común (Windows, Linux y Mac OS X). Ha impartido cursos de formación de sistemas operativos y seguridad para empresas y administraciones públicas, y desde 2003 imparte docencia universitaria en diversas titulaciones oficiales de Ingeniería Informática e Ingeniería de Telecomunicación, en asignaturas de sistemas operativos, sistemas distribuidos, concurrencia y seguridad. Es autor de múltiples publicaciones científicas en conferencias como *IEEE PerCom*, *Bell Labs Technical Conference* o *International Workshop on Plan 9*, y revistas como *IEEE Pervasive Computing*, *Pervasive and Mobile Computing*, *Journal of Systems and Software*, *Concurrency Computation Practice and Experience*, *Software Practice and Experience*, *Computers & Security* o *Bell Labs Technical Journal*.

Contacto:
`gorka.guardiola@urjc.es`
https://paurea.net/
`twitter: @paurea`