Sistemas operativos

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

8 de febrero de 2021

Índice de figuras

Índice general

Lista de Programas

Sobre este libro

Este es un libro de sistemas operativos. Lo estamos escribiendo durante la cuarentena del COVID-19 como parte del material online para dar soporte a los alumnos. Como tal es un libro eminentemente práctico, aunque hay capítulos con conceptos más teóricos. Esos capítulos se pueden complementar con otras referencias bibliográficas clásicas (ver por ejemplo [1, 2, 3]). Los ejemplos los hemos probado en Linux, pero en principio deberían ser fáciles de trasladar a cualquier sistema tipo Unix.

1 Introducción y estructura del sistema

1.1. Historia

En la primera era de los computadores digitales (≈ 1945-1955), los ordenadores eran del tamaño de una sala completa. Al principio, esas computadoras se construían con relevadores electromecánicos, 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 (≈ 1955-1965) 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 (≈ 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*). Estos sistemas permitían a varios usuarios acceder al sistema y utilizarlo de forma concurrente. 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 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*[1].

En esa época había dos familias diferenciadas de sistemas Unix: los sistemas de la Universidad de Berkeley, *BSD (Berkeley Software Distribution)* (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] 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, a partir de ahora, usaremos la segunda forma genérica.

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 el actualmente Windows 10 incorpora un Linux virtualizado llamado Windows Linux Subsystem).

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.

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

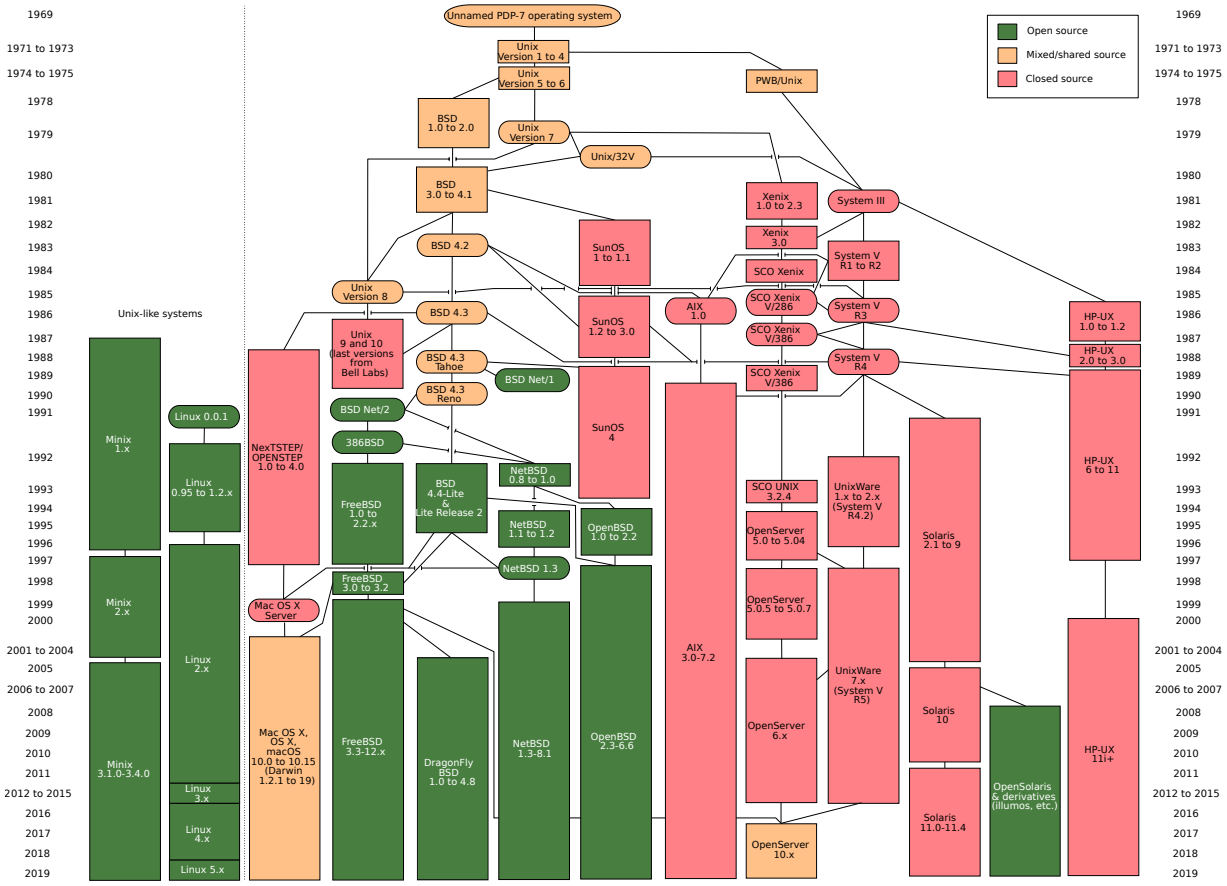

Imagen (cc) Wikipedia. Autores: Eraserhead1, Infinity0, Sav_vas

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

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 que no existe. Oculta los detalles de la máquina y la complejidad que supone compartir la máquina entre distintos programas en ejecución. De esta forma, podemos escribir nuestros programas sin estar pendientes de cómo se reparten los recursos (CPU, 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. Es un programa, normalmente escrito en C y en ensamblador. No es magia. Es un

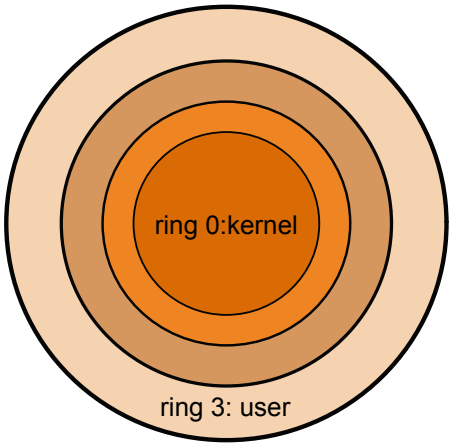

Figura 1.5: Tradicionalmente sólo se usan dos anillos de privilegio de la CPU: ring 0 para el kernel y ring 3 para los programas de usuario.

programa que es 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 se pueden usar las instrucciones para gestionar los recursos.
- Ring 0: núcleo del sistema, se pueden 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 las instrucciones que ejecuta el *firmware* para la gestión de energía, etc. (SMM o System Management Mode) y -3 para el llamado Intel ME (sin documentar, procesador propio que inicializa el procesador principal y monitoriza el sistema).

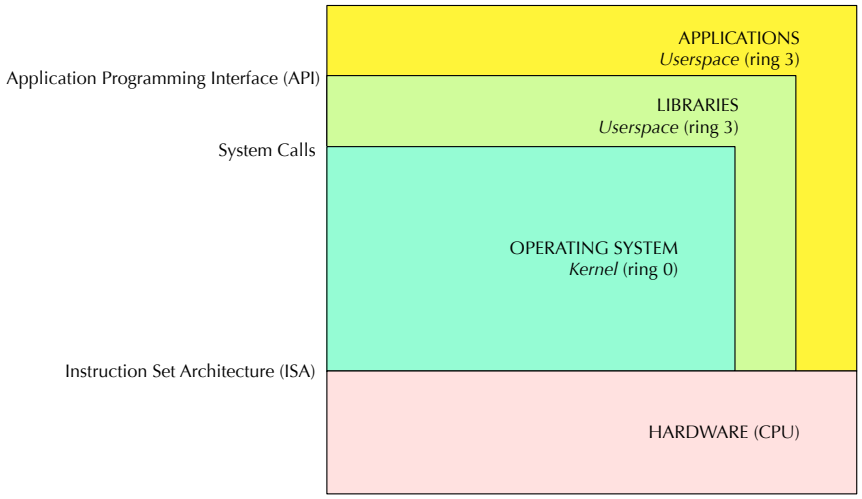

Figura 1.6: Estructura del sistema.

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.

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 (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 muchas de las herramientas y aplicaciones. Por ejemplo, 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 *Linux Standard Base*, *LSB*.

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 del sistema operativo ejecuta en **modo privilegiado** (ring 0). Esto significa que puede ejecutar instrucciones especiales (acceder a ciertos registros de la CPU, invalidar caches, 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 (concurrencia) 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 lo son.

La mayoría de los kernels actuales permiten la carga dinámica de *módulos* para ampliar/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. En cada sistema operativo los módulos tienen una extensión diferente: en Linux es `.ko`, en Mac OSX es `.kext`, en FreeBSD es `.kld`, en Windows es `.sys`, etc.

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

- `lsmod` escribe en su salida la lista de módulos cargados actualmente.
- `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 `/lib/modules/`*`versión-de-kernel`*`/`[2]:

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

Por ejemplo:

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

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

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 hardaware, flujos de control, comunicación y gestión de memoria. El resto (lo que se llama *OS personality*: gestión de procesos, red, sistemas de ficheros...) se implementa en servidores independientes. La idea fundamental es separar las **políticas** en espacio de usuario y dejar los **mecanismos** en espacio de kernel. Las ventajas son la modularidad y la tolerancia a fallos en los componentes. El problema es que es complicado que sean tan eficientes como los monolíticos por la comunicación entre sus componentes. Los primeros microkernels eran 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 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

Todos los demás programas son programas de usuario. Por ejemplo, las aplicaciones que utilizamos ejecutan en área de usuario. Otros programas de usuario son herramientas del propio sistema operativo, como el *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.

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

En general, estos son los dos elementos fundamentales para un flujo de ejecución (o de control):

- **Contador de programa**: un registro de la CPU[3](PC, *Program Counter*) que apunta a la instrucción por la que va ejecutando el programa.
- **Pila**: un registro de la CPU (SP, *Stack Pointer*) que apunta a zona de la memoria donde se guardan los **registros de activación** (o marcos de pila) donde se guardan los datos necesarios para realizar llamadas a procedimiento (parámetros, variables locales, dirección del programa para retornar, etc.).

Estos son los dos datos básicos que forman parte del **contexto de un 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 pueda 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 (pseudo-paralelismo). En cualquier caso, el sistema operativo crea la ilusión de que cada proceso tiene su propia CPU. Para el programador, no hay diferencia entre estos dos casos.

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. 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 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[4] en la arquitectura AMD64:

1. Coloca los argumentos en la pila del proceso.
2. Carga el número de llamada al sistema en un registro.

[3]El registro puede tener distinto nombre dependiendo de la arquitectura, por ejemplo en Intel de 64 bits este registro se llama `RIP`.

[4]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 similar a Unix, tiene importantes diferencias.

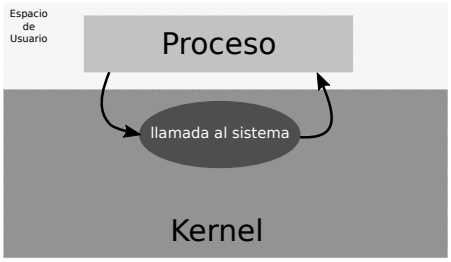

Figura 1.7: Llamada al sistema

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, el puntero de pila de área de usuario y el contador de programa por el que está 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 llama a `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 hay que inicializar antes 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 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, salta a una posición de memoria concreta. En esa posición suele haber proyectado en memoria un programa que viene almacenado en una

ROM o flash por el fabricante de la placa madre: el *firmware*. Ese programa básicamente inicializa el hardware y carga el cargador primario. Un ejemplo de ésto es *EFI* en un PC[5] o *U-boot* en muchos dispositivos con procesador ARM.

2. El cargador primario seguramente cargue otros cargadores secundarios. El objetivo final de la cadena de cargadores es inicializar el hardware lo suficiente 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, de un disco óptico (DVD) o de un disco externo. Ejemplos de cargadores (incluyen primario y secundario) son *grub* y *LILO*. 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. Se inicializa el núcleo (varias fases distintas), se configura el sistema de ficheros princial, etc. Al final se crean los primeros procesos de usuario, entre ellos *init*.

4. Ese programa *init* va creando los procesos para ejecutar los programas de usuario (shells, interfaz gráfica, servidores, etc.)[6]. 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 ejecutar 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*:

- **Máquina virtual de proceso**: tiene como objetivo proporcionar una plataforma para ejecutar un único programa: emuladores de otra ISA[7] (p. ej. OSX Rosetta), optimizadores, ISA 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. El trabajo de un hipervisor es muy parecido al de un kernel: multiplexar la máquina (pero para usar distintos sistemas encima). El hipervisor proporciona un entorno completo y persistente para ejecutar un sistema operativo completo.

[5] Antes este firmware se llamaba BIOS. En los PCs modernos, el firmware es UEFI, y la forma en la que está programada es diferente.
[6] En los sistemas GNU/Linux actuales, `systemd` es el proceso que hace el papel de init.
[7] La ISA, Instruction Set Architecture es el conjunto de instrucciones de un procesador.

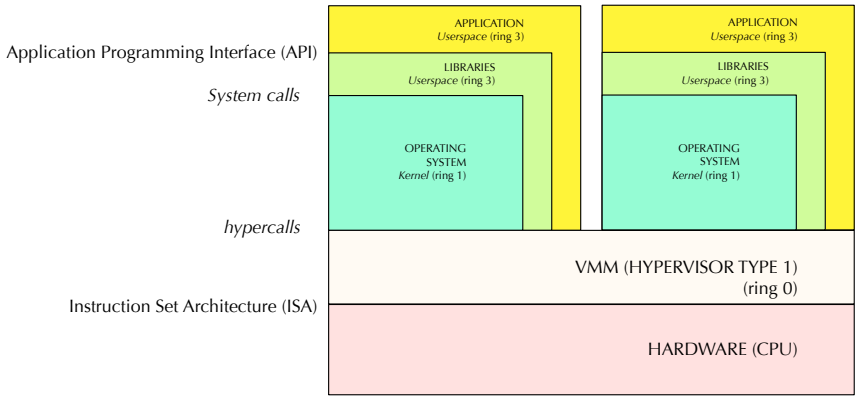

Figura 1.8: Estructura con paravirtualización.

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.

Máquina virtual clásica (hypervisor type 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 HW, etc.

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

- **Virtualización completa asistida por hardware**. Se basa 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 type 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.

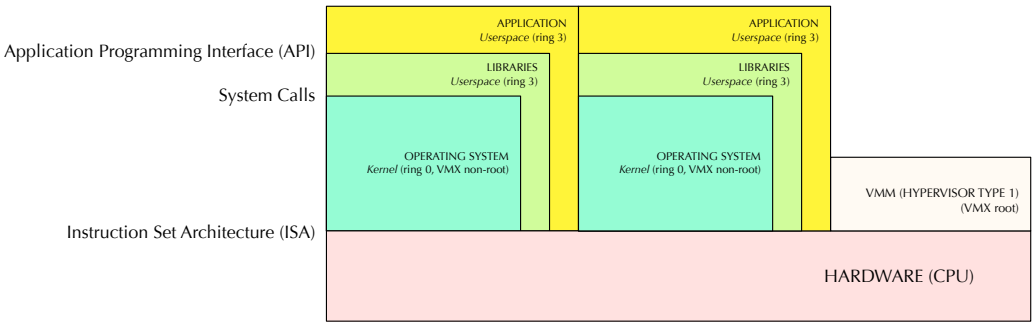

Figura 1.9: Estructura con virtualización asistida por hardware.

Si se habla de una *Whole-system VM* también se emula la ISA de otro procesador. Virtual PC es un ejemplo.

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 necesitas 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. 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.

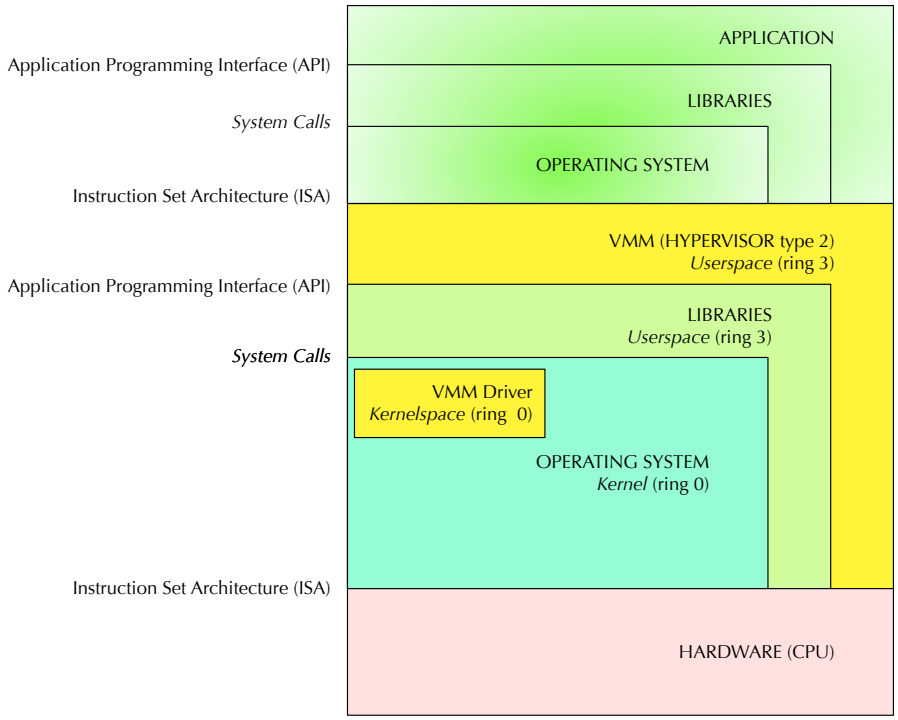

Figura 1.10: Estructura con máquina virtual alojada.

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:

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.

El Capítulo 2 está dedicado entero al uso de la shell.

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, tienes que introducir el nombre de usuario y la contraseña asociada. El sistema autenticará al usuario y abrirá una sesión. Entonces ya podremos 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

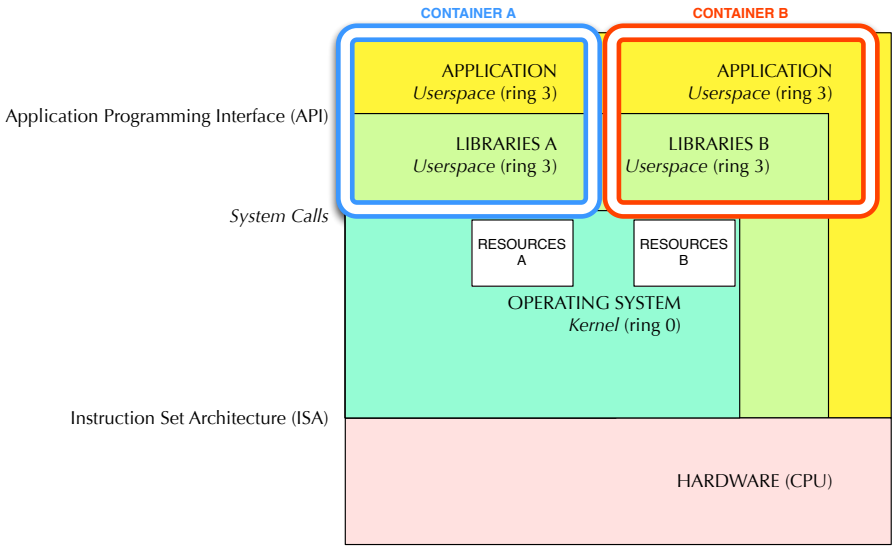

Figura 1.11: Estructura con contenedores.

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*).

1.4.1. Árbol de ficheros

Los datos están organizados en árboles 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, 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.

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.

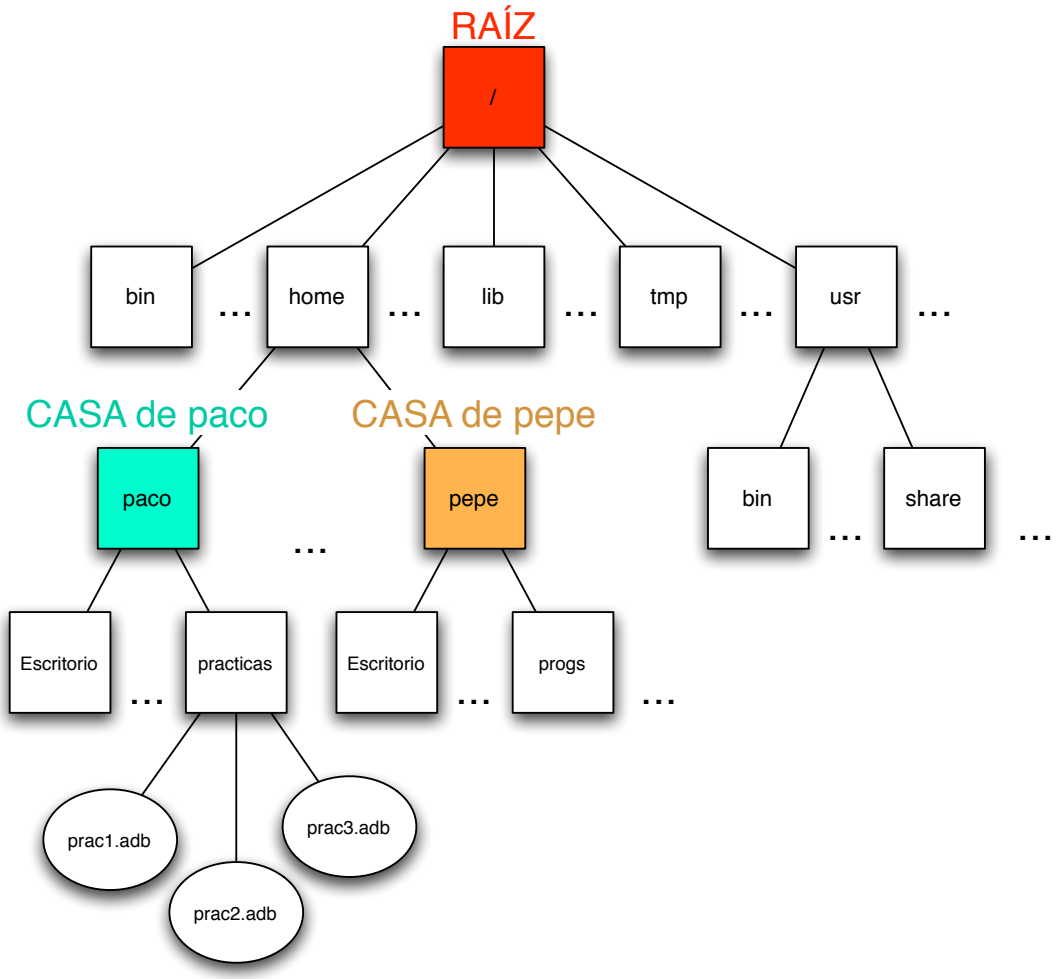

Figura 1.12: Árbol de ficheros.

- `/home` tiene los datos personales de los usuarios.
- `/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 (cache, logs y otros ficheros que generan los programas).
- `/boot` contiene los ficheros de arranque del sistema (cargadores, núcleo, etc.).
- `/media` y `/mnt` 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`[8]:

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

El directorio de trabajo en una shell recien 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:

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

```
 $ 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
 $
```

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
```

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

```
 $ man 3 printf
```

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 consul-

tamos. 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. Se puede ver la página de manual para ver la tabla de caracteres y su representación numérica.

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

- 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 caracter 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 para poder leerlo.

- UTF-8: puede usar 1, 2 o más bytes. Es compatible hacia atrás con ASCII y permite representar los caracteres de todos los alfabetos del mundo. Podemos ver llos 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)*

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. Es un convenio usado en todos los programas, bibliotecas, etc. en Unix. 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):

- **: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 Unix. Los principiantes deben 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.
- `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 el las primeras/últimas líneas del fichero en su salida.
- `tar`: crea un fichero con múltiples ficheros dentro (comprimidos o no).
- `gzip/gunzip`: comprime/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). Esos procesos tienen su propia copia de la variable, 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 comado `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` deja buscar comandos que ejecutamos hace tiempo[9].
- `Ctrl+c` mata el programa que se está ejecutando.
- `Ctrl+z` detiene el programa que se está ejecutando.
- `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.

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

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 [5].

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 todos los sistemas vienen con una interfaz gráfica, sigue siendo importante aprender la shell, a pesar de resultar más complicada, ¿por qué? La razón es que 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 el desazón que acompaña a estas tareas. ¡Estás realizando el trabajo de la máquina!

El nombre de shell (o 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.

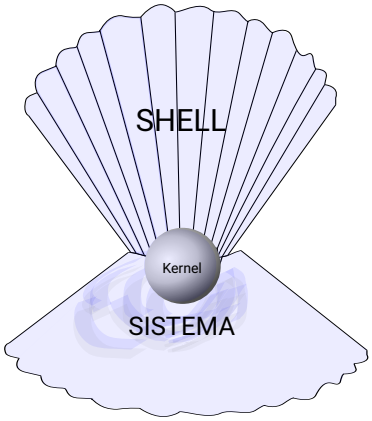

Figura 2.1: Explicación del nombre shell

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 los ejecutará. Cuando la shell ejecuta de este modo, recibe el nombre de shell

[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.

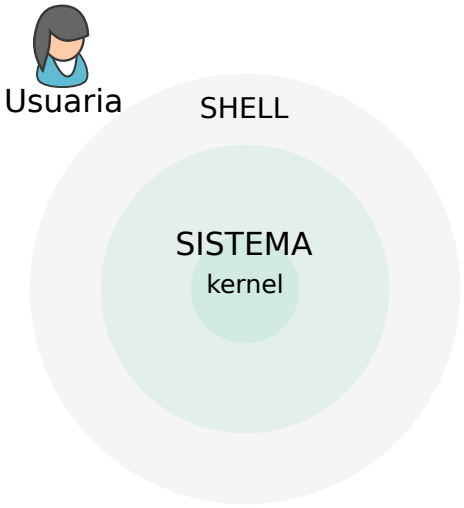

Figura 2.2: Relación más precisa de la shell con el sistema

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.

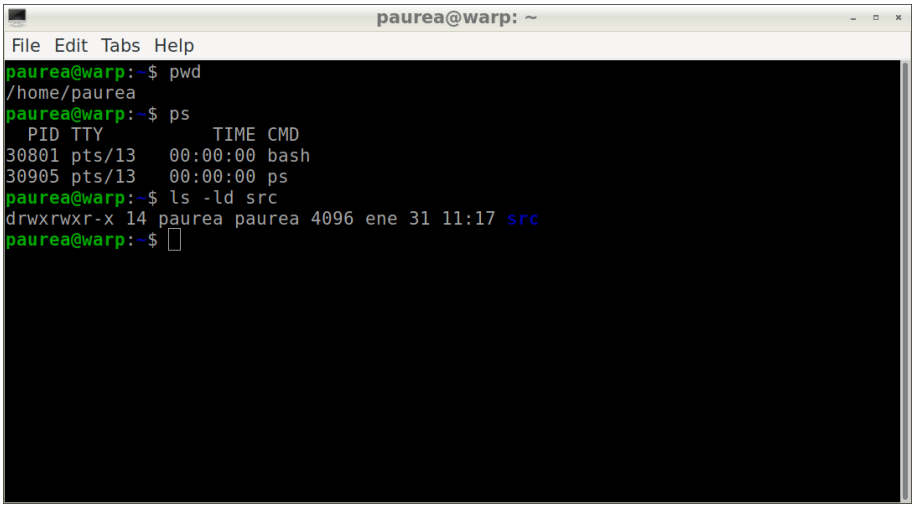

Figura 2.3: Un terminal con una shell y varios comandos

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-

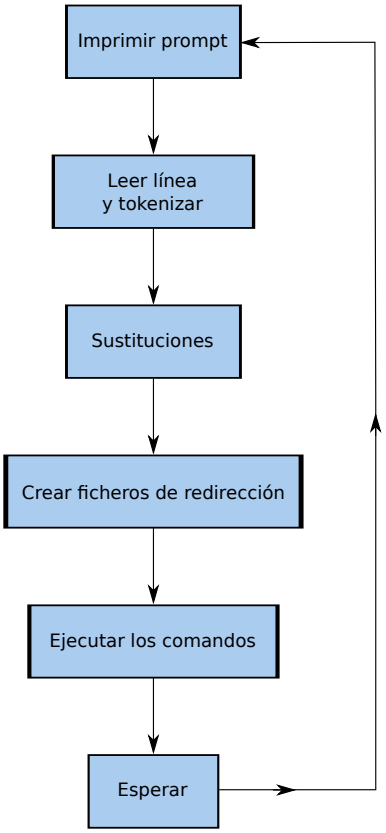

Figura 2.4: Pasos que realiza una shell interactiva

gramador, es el libro clásico [6]. Este libro puede darnos una idea de la filosofía original del Unix original de combinar pequeñas utilidades mediante la shell. Una versión más avanzada y moderna de este libro podría ser [7], aunque este libro 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 PC 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**. Todos los comandos que se explican aquí deberían funcionar en cualquier terminal de Unix sin problemas.

2.2. Configuración

La shell trabaja principalmente con texto. Hay partes de la shell, de procesado de texto, como las expresiones regulares y globbing (ver la sección 2.21 para más detalles) cuyo funcionamiento depende de las *locales*, es decir, de la configuración del idioma, país, etc. Para evitar estas dependencias a la hora de programar, tener el manual en versión original (en inglés, que tiene menos errores) y un funcionamiento constante de la shell,

se recomienda tener la variable de *entorno* `LANG` puesta a `C`. Explicaremos la razón de que esto funcione en la sección 2.8.1. Si se quiere hacer de forma no persistente, basta con escribir `export LANG=C` en una ventana (y estará puesto en esa ventana).

Si se quiere hacer de forma más persistente, hay que cambiar un fichero de configuración para la shell interactiva por defecto.[3] Ésta en Ubuntu es bash. Para ello se puede ejecutar lo siguiente (copia con cuidado los dos comandos), `cd` y `echo "LANG=C" >> .bashrc` con las comillas dobles, el igual y los dos mayores[4]:

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

De nuevo, la razón del funcionamiento de esto debería quedar clara tras entender el funcionamiento de la shell.

2.3. 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`. 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. 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`[5].

Algunos comandos tienen modificadores que no empiezan por guión e incluso el mismo con guión y sin guión que significan cosas diferentes. Esto es debido a que heredan esta sintaxis sin guión de `BSD`. El caso más famoso es `ps`, el comando `ps -a`, y `ps a`, hacen cosas diferentes.

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*). El modificador `-v` (*verbose*)suele modificar como de ruidoso o dicharachero es el comando, es decir cuanta información escribe sobre lo que hace a su salida. En cualquier caso, es conveniente ver la página de manual concreta del comando.

[3] De esta forma las shells que ejecuten un script la heredarán en el *entorno*, como se verá en la sección 2.8.1.

[4] En los ejemplos de shell del libro representaremos el prompt como `$ `.

[5] 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)`.

A veces es interesante que un argumento 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`.

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

2.4. 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.[6]

Un fichero en Unix es un programa interpretado mediante un mecanismo basado en lo que se llama el número mágico. Cuando se ejecuta un programa, la llamada al sistema exec, véase `execv(3)`, recibe un nombre de fichero a ejecutar. Los primeros bytes de este fichero se utilizarán para decidir qué se va a hacer en la ejecución y reciben el nombre de número mágico. 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 fichero interpretado y se ejecutará el intérprete que viene a continuación con sus argumentos (el resto de la línea) pasando como último argumento el nombre 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)]^{7}. 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. 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. El comando `chmod` sirve para dar permisos (en este caso de ejecución) al programa. Veremos más detalles sobre esto más adelante.

Ejemplo, un script de shell.

Script 2.1: script

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

[6] 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 librería de C `isatty(3)`.

[7] 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 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
 $
```

Ejemplo, ejectuar cat como intérprete:

Script 2.2: quine

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

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

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

Ejemplo, ejecutuar 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 formas es escribir scripts de shell. Un script es un programa interpretado mediante el mecanismo

hash bang que utiliza la shell como intérprete. Para depurar estos programas se puede poner `-x` como parámetro para que la shell escriba el comando antes de ejecutarlo.

Script 2.4: hola.sh

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

También se pueden poner líneas con `echo` para trazar el programa como con cualquier otro 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.

¿Cómo decido cuando tengo que realizar una tarea si utilizar un lenguaje de programación (C, Java, Go, $C++$), escribir un script de shell o símplemente 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...

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 cuanto se tarda en ejecutar la tarea manualmente, cuantas veces se realiza y cuanto cuesta en esfuerzo mental la automatización. En este coste hay que contar el tiempo en 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 un gran tiempo y 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, generalidad, en ese orden [8].

2.5. 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[8]. 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 o *foreground* y si no, se dirá que ejecuta en segundo plano o *background*. La variable de shell especial $! contiene el el `pid` del último proceso que ejecutó en segundo plano.

```
 $ 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*, lo que 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
```

[8] Esto se corresponde con que la shell llame a `wait(2)` para los procesos hijo que derivan de esa línea de comando

2.6. Diferentes shells

Hay muchas shells diferentes. `sh` es la shell original de Unix, escrita por Ken Thompson. Fue reescrito por Stephen Bourne en 1979 para Unix Version 7: *bourne shell*. Los sistemas derivados incluyen distintas shells: `sh, ash, bash`[9]`, 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 a su shell por omisión para ejecutar scripts que suele comprobar se ejecuta con el nombre `sh` y comportarse de forma más estricta. En Ubuntu y Debian es `dash`. No hay que confundir ésta con la shell interactiva por omisión para un usuario que 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 [9].

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 tienen la mayoría de las shells.

Un buen libro para aprender a programar scripts de shell es [10].

2.7. Comandos internos o builtins

Hay comandos que se implementan dentro de la shell (no se ejecutan un fichero externo al shell, sino que son parte de la propio 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 es estado del último comando que ejecutó (el valor de `$?` como veremos en la sección 2.10).

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
```

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

2.8. 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 lo llamados *caracteres de escape*. Son tres, la barra invertida o *backslash* que es el caracter '\', las comillas simples 'texto'.[10] 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 3, el dólar (que afecta a lo que viene 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 $
 $
```

2.8.1. Variables

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 shell tras leer la línea, como se puede ver en la figura.

```
 $ x=bla
 $ echo $x
bla
 $ touch c d e
 $ ls
c  d  e
 $ cmd=ls
 $ $cmd
c  d  e
 $
```

[10] 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 el ejempo anterior, el comando `echo` no sabe nada de $x, es la shell la que sustituye el texto y ejecuta la línea `echo bla`.

Es importante entender que esto las define en la shell, pero no las exporta a los hijos. Para esto 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 de shell. El comando builtin `export` sirve para meter a una variable de shell en el entorno y que se herede 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 en Unix, además, pueden consultar estas variables aunque no estén escritos en shell[11] y cambiar su funcionamiento. Esto es bastante común y es una forma de configurar el sistema.

2.8.2. 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:

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

[11] En siguientes capítulos veremos cómo hacerlo con `getenv(3)`.

2.9. Redirecciones

2.9.1. Entrada estándar, salida estándar y redirecciones

En Unix, los ficheros abiertos de un proceso están numerados (File Descriptor, FD o *descriptor de fichero*). Cada vez que un proceso llama a `open(3)` sobre un path para abrir un fichero o directorio, 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 defecto: entrada y salida estándar y de error, 0 (**stdin**), 1 (**stdout**), 2 (**stderr**). Si no se realiza ninguna redirección, estos ficheros se corresponden con dispositivos en los que leer o escribir interactua con la consola, es decir, se imprime algo por el terminal o se espera entrada del usuario, como se puede ver en la figura 2.5.

- Stdin es de donde lee datos por defecto el programa (entrada estándar).
- Stdout es dónde escribe datos (salida estándar).
- Stderr es dónde escribe mensajes de error (salida de error).

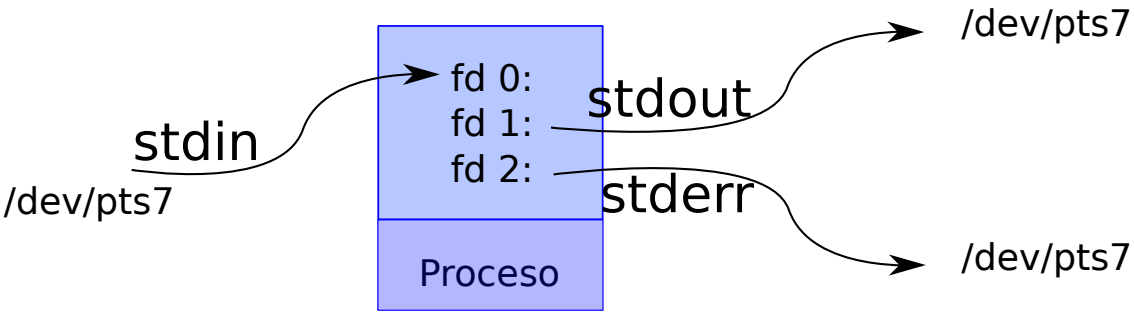

Figura 2.5: Entradas y salidas de un proceso

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 o dársela a otro programa, etc. La razón para tener la salida de error separada es que de esta forma se pueda mandar la salida de un programa a otro sin que los mensajes de error interfieran. Los detalles de cómo funciona la redirección en la shell se puede ver en `dash(1)`, ver la sección *Redirections*.

Para redireccionar la entrada y la salida estándar de un comando desde un fichero o a un fichero, se utilizan los caracteres mayor y menor, <' y '>'. Estos caracteres hacen que el proceso abra los ficheros de redirección antes de ejecutar . En el caso del fichero de salida, lo crea si no existe y si ya existe lo trunca, 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.

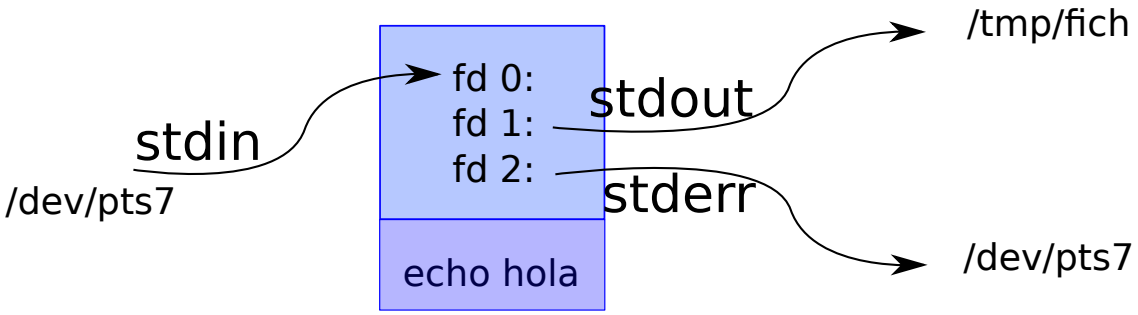

Figura 2.6: Ejecución con la salida estándar apuntando a fich

Para redireccionar la salida de error '2>' (en general, para entrada o salida con un número delante, se redirecciona ese descriptor de fichero p. ej. '5<')

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

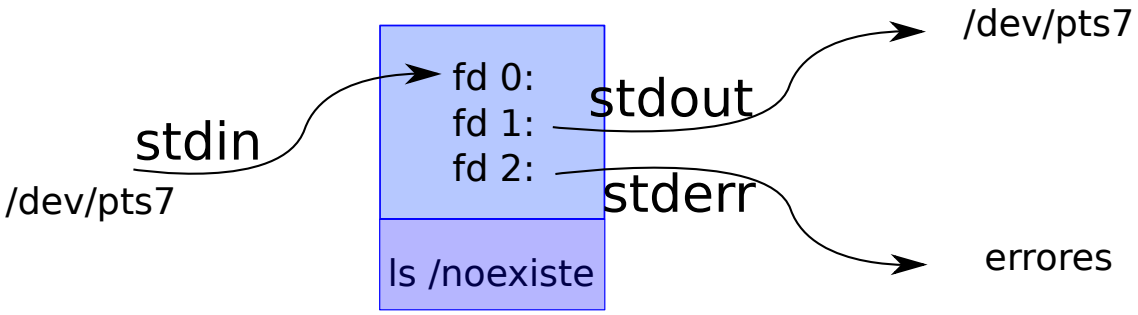

Figura 2.7: Uso de la salida de error

Si no queremos que se trunque el fichero que ponemos a la salida, '>>' hace lo mismo que '>' pero abre el fichero en modo *append*. 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
 $
```

Ojo, un error común es ejecutar un comando y especificar el mismo fichero que se le pasa como argumento o como redirección de entrada como salida:

```
 $ 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.

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.

Ojo, el orden importa y se interpreta de izquierda a derecha. Esto redirecciona a fich la salida estándar y luego la de error 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
 $
```

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
 $
```

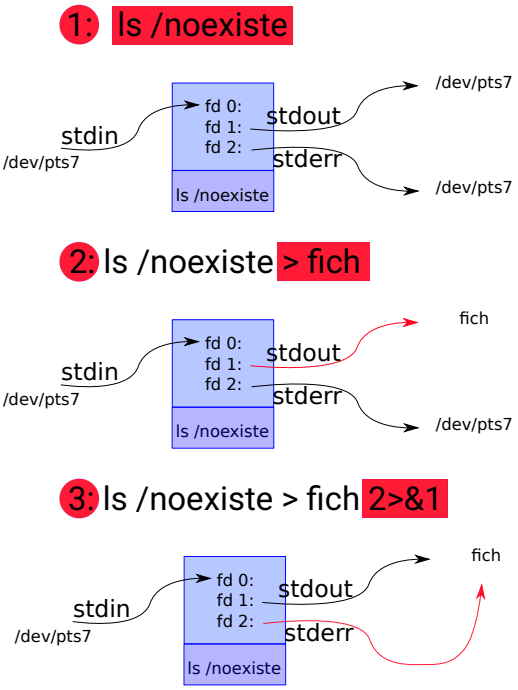

Figura 2.8: Redirección a fich de la salida estándar y la de error

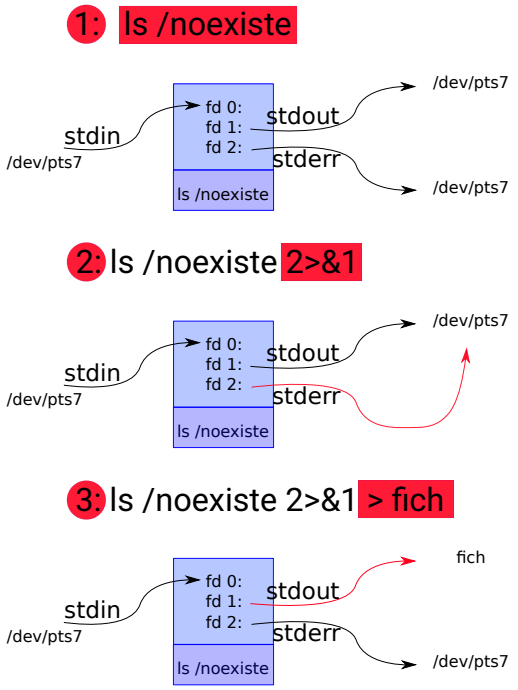

Figura 2.9: Redirección incorrecta a fich de la salida de error

Para escribir algo por la salida de error de un script, es decir, que el comando echo escriba en lo que estaba conectado en su salida estándar se redirecciona la salida estándar a la de error:

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

Así el script:

Script 2.6: aaaerror.sh

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

Escribe por su salida de error:

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

2.9.2. Ficheros especiales

Hay una serie de ficheros especiales que sirve el kernel cuya función es generar datos o recibir datos y hacer algo con ellos. 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 abierto 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.

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. El fichero `/dev/zero` es una *fuente* de bytes a cero, cualquier lectura retornará bytes a cero.

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 del proceso en ejecución se puede ver en la figura 2.10.

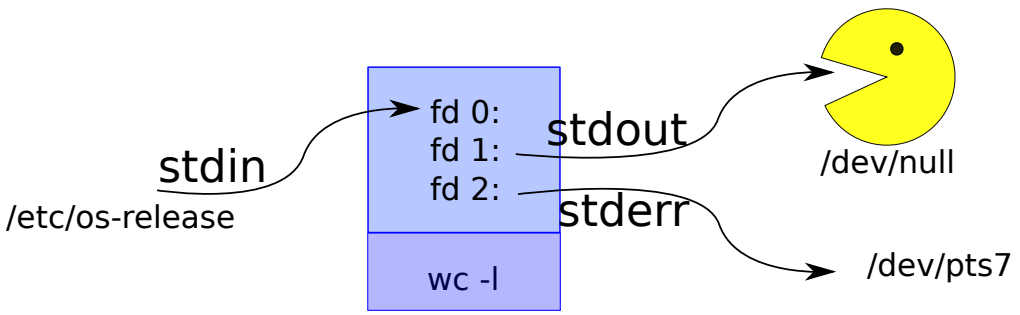

Figura 2.10: Hacer callar a un proceso mediante /dev/null

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.

Este comando genera un fichero con 1kB de ceros.

```
 $ 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 y desde dónde, etc.

Un truco avanzado para crear un fichero de 10MB extremadamente rápido (y si tenemos suerte, el fichero sólo ocupa un byte) 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` o más sofisticado para ficheros de texto (partir por tamaño o por expresiones regulares del contenido es `csplit`).

2.9.3. 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 se 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 lista los ficheros de `/tmp` y cuenta cuantos hay (`wc -l` cuenta las líneas de texto que hay en su entrada).

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

Los procesos en ejecución se muestran en la figura 2.11.

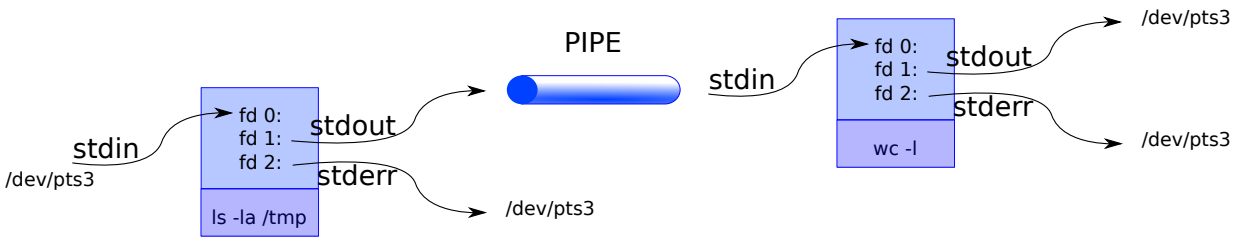

Figura 2.11: Conexión de dos procesos mediante pipes

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

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

En ocasiones es interesante ver el resultado de un programa y escribirlo en un fichero simultáneamente. Esto es lo que hacer el comando `tee`.

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

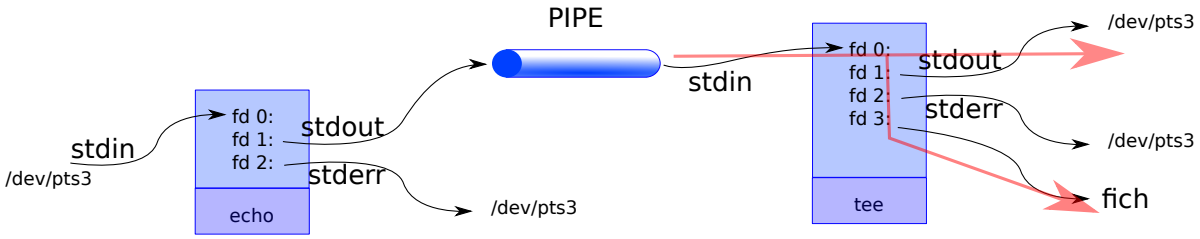

Figura 2.12: Ejemplo de uso del comando tee

Esto 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
 $
```

2.10. Estado de salida (status)

Cuando un comando sale, le deja al proceso padre su estado de salida (*status*), que es el argumento de la llamada al sistema `exit` (ver `exit(3)`). El estado del último comando que se ejecutó (o *pipeline*) se guarda en la variable de shell `$?` y es un número. Si ese número es 0, el comando ha tenido éxito, en otro caso, ha habido un error. Se puede (y se debe) hacer salir a un script con el 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) de si ha habido un error en su ejecución.

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 y false mediante el de error. 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 (y que se utilizan también por sus propiedades de cortocircuito). La línea de comando `cmd1 && cmd2` ejecuta el segundo comando sólo si el primero ha tenido éxito (cortocircuito). Como consecuencia, la línea de comando `cmd1 && cmd2` sólo tiene éxito si ambas líneas tienen éxito (*and* del álgrebra de Bool). De forma similar `cmd1 || cmd2` ejecuta el segundo comando sólo si el primero 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
```

2.10.1. Test

El comando `test` sirve para comprobar condiciones de distinto tipo que devuelve el comando en forma de estado de éxito o de fallo al salir.

Por ejemplo, para ver si existe un fichero:

```
 $ 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
 $
```

Condiciones sobre ficheros:

- -f fichero
  si existe el fichero.
- -d dir
  si existe el directorio.

Condiciones sobre cadenas:

- -n String1
  si la longitud de la string no es cero.
- -z String1
  si la longitud de la string es cero.

- String1 = String2
  si son iguales.
- String1 != String2
  String1 and String2 variables no son idénticas.
- String1
  si la string no es nula.

Condiciones sobre enteros:

- Integer1 -eq Integer2
  si los enteros Integer1 e Integer2 son iguales.
- -ne: not equal.
- -gt: greater than.
- -ge: greater or equal.
- -lt: less than.
- -le: less or equal.

Hay dos comando `test` instalados en el sistema, que pueden ser el mismo, `test` y [. 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, tengan un aspecto normal. Esto: `[ $a -eq $b ]` es lo mismo que `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 corchete, mira que el último argumento también lo 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* que se con el mismo nombre u otras cosas (por ejemplo, puede ser un alias o una función). Para saber qué es lo que se va a ejecutar realmente y de qué tipo es, tenemos el *builtin* `type`.

Así vemos que, en las shells modernas, test es normalmente un builtin.

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

Si quiero ejecutar un binario concreto en lugar del builtin, puedo utilizar el path absoluto.

2.11. Parámetros posicionales

Un script puede acceder a los parámetros posicionales que se le han pasado al ejecutarlo. Se pueden acceder a estos parámetros que se han pasado al script mediante las variables especiales de shell $1, $2, $3... y $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 $* contiene los parámetros posicionales. La variable "$*" 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 2.14 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). Se 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 2.14), 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 es que se extraen los parámetro optativos de la lista de argumentos y así el resto del código no necesita ser condicional.

2.12. 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; ...)` o ejecucion en el mismo 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 actual (`cd, export`); Por ejemplo:

```
#sigo en tmp al final:
 $ pwd; (cd /etc; ls apt;); pwd;
/tmp
apt.conf.d  sources.list
preferences.d  sources.list.d  trusted.gpg  trusted.gpg.d
/tmp
#BLA no existe al final:
 $ echo z$BLA; (export BLA=bla; echo $BLA;); echo z$BLA;
z
bla
z
 $
```

2.13. Otras sustituciones

2.13.1. Here documents

Los documentos en línea (*here documents*) están a mitad de camino entre una sustitución y una redirección. Se especifica que un trozo de texto lo que va a recibir un comando como si se lo hubiesen pasado a su entrada con una redirección. El trozo de texto se especifica desde '<<MARCA' hasta una línea que tiene 'MARCA'. Ejemplo:

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

Nótese que `cat` no ha recibido argumentos, es una redirección a la entrada:

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

2.13.2. 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 se utiliza, sobre todo, para encajar nombres de ficheros y directorios (aunque no sólo para ésto).

Las expresiones regulares son un lenguaje más general que está 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`) 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.

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, se dejan tal cual, a diferencia de las variables, que si no existen se sustituyen la cadena vacía).

```
 $ 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
 $
```

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 (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
 $
```

2.14. 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
```

Tiene como condiciones los resultados de ejecutar comandos. Un ejemplo de `if` utilizando test y la construcción && se puede ver a continuación. Si el fichero de NOTAS existe y el script ha recibido `-c` como primer argumento, escribe por su salida de error un error y sale con estado de error. Si no, calcula la notas, crea el fichero si se lo han pedido y añade las notas al final. Ojo a 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 (ojo, es una 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. Ojo, todas las estructuras de control se pueden escribir separando sus partes (cabecera del if, etc.) mediante un fin de línea o mediante un punto y coma ';'. 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

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

El case puede contener patrones de globbing que se encajarán con la palabra (cuidado, no se encajan con nombre 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
```

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
```

```
 $ ./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. El `while` es similar al if (pero iterando):

```
while comando
do
        comandos
done
```

Por ejemplo, esto es un reloj:

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

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:

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

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.

El comando `seq` recibe el parámetro `-w` para mantener el ancho de los números constantes

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

2.15. 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
 $
```

Esto itera 2 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.

En comparación, este 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
 $
```

2.16. Variable IFS

Esta variable 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
 $
```

2.17. 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 (ojo, los oculta, hace ocultación o *shadowing*).

Por ejemplo:

```
#!/bin/sh

hello () {
        echo hola $1
        shift
        echo adios $1
}

hello uno dos $1
 $
```

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

2.18. Alias

Son similares a las funciones pero más sencillos. Pensados para definir nombres cortos para la shell interactiva para comandos con argumentos que se ejecutan mucho. Es un tipo de sustitución de la shell. Sin argumentos, `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'
 $
```

2.19. Operaciones aritméticas

Para operaciones básicas con enteros podemos utilizar el propio shell. También podemos usar el comando `bc`. Una expresión tras un dólar y con dobles paréntesis, `$((arithexpr))` se sustituye por la evaluación de la expresión. Por ejemplo `$((5 + 7))` se reemplaza por 12.

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

2.20. 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 verdadero fallo (argumentos erróneos, ficheros que no existen...), salen con estado 2. El comando `diff` saca por su salida un conjunto de ediciones para convertir un fichero en el otro. La salida de `diff` se puede aplicar mediante el comando `patch` para aplicar estas ediciones en un fichero y convertirlo en el otro[12].

```
 $ 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
 $
```

El comando `diff` se puede aplicar de forma recursiva para comparar dos árboles de ficheros con el parámetro `-r`.

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

2.21. Filtros y expresiones regulares

2.21.1. 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 caracters 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.

```
 $ 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 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:

```
 $ touch fich
 $ cat fich
 $ (sleep 5; date >> fich; sleep 10; date >> fich) &
[1] 7385
 $ tail -f fich
vie abr 17 20:33:57 CEST 2020
```

Pasados 10 segundos:

```
 $ touch fich
 $ cat fich
 $ (sleep 5; date >> fich; sleep 10; date >> fich) &
[1] 7385
 $ tail -f fich
vie abr 17 20:33:57 CEST 2020
vie abr 17 20:34:07 CEST 2020
```

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

El comando sort puede recibir una lista de columnas (empezando por la 1) y un separador. Ordena cada línea usando esos campos como clave (es un intervalo de campos).

```
 $ 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
 $
```

Ojo, sin el parámetro `-s` el sort que realiza no es estable. Un sort 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.

2.21.2. 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 que se pueden definir mediante expresiones regulares[13]. Se dice que una cadena de texto encaja con la expresión regular si pertenece al lenguaje que describe. Por ejemplo el lenguaje correspondiente a $L = \hat{}a*b\$$, que es $L =$ {'b', 'ab' , 'aab', 'aaab', 'aaaab',... }.

Mediante una expresión regular[14], se define un lenguaje regular para describir/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 `regex(7)`.

Las expresiones regulares no sólo están en la shell. Hay muchos motores de expresiones regulares y dialectos (Perl, Python...). Se utilizan en IDEs y editores, en motores de

[13] Los lenguajes regulares pueden describir 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 [11]

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

búsqueda y en muchas aplicaciones. Un buen libro para aprender expresiones regulares es [12]. Uno más avanzado es [13].

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 son:

- Paréntesis para agrupar (precedencia de operadores) (e_1).
- Operador alternativa $e_1|e_2$.

  Por ejemplo, para dos expresiones a y b $a|b$ significa que la expresión puede encajar con el primero `'a'` o el segundo, `'b'`.
- Operador escape \ para que un carácter pierda su significado especial (escaparlo dentro de la expresión regular), por ejemplo \(es el caracter paréntesis `'('`.
- Operador concatenación (no se escribe, como la multiplicación), sencillamente juxtapuesto.

  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.
- Conjuntos de operadores: Cualquier carácter, el punto: . El conjunto, enumeración de rangos entre corchetes: $[a-c]$ o de caracteres sueltos $[acd]$ y 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 y `[^abc]` *no* encaja con un carácter que no puede ser `'a'` ni `'b'` ni `'c'` , sin embargo sí encaja con `'z'`.
- 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.
- Operadores de repetición de Kleene, $*$, $+$ y el opcional ?. La expresión e_1* encaja si aparece *cero o más veces* la regexp que lo precede. La expresión e_1+ encaja si aparece *una o más veces* la regexp que lo precede. 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`$+$.

Algunos operadores y construcciones son redundantes:

- 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 `'aaab'` o `'aaab'` o `'aaab'`, va a encajar siempre con la subcadena más larga, es decir: *aaab*. Para limitar que parte de la subcadena encaja se pueden utilizar delimitadores como ^ que dicen que encaja con el principio de la cadena (en un comando orientado a líneas, 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 importante que la expresión regular sea precisa, es decir, que encaje con las cadenas que buscamos y no con otras.

Veamos dos ejemplos:

- El lenguaje (finito) definido por $L =$ ^(hola|adios)[12a-c]$ es

  $L =$ {'hola1', 'hola2', 'holaa', 'holab', 'holac', 'adios1', 'adios2', 'adiosa', 'adiosb', 'adiosc'}. La cadena 'holaadios' no encajaría con la ésta expresión regular, ya que lo impide el delimitador.

- El lenguaje (infinito) definido por $L =$bueno|((malo)*)$ define el lenguaje

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

  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.

2.21.3. Grep

El comando `grep`[15] 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, metiendo líneas en su entrada estándar para que escriba por su salida estándar las que encajen.

Hay tres comandos de la familia de `grep`:

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

- El comando `fgrep`, que busca texto literal. 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`.

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

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 de `egrep` son:

- `-v`: Realiza lo inverso: 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 status de salida).

Este comando se utiliza tanto para ver si la string está o no está en un fichero (con el estado de salida), como para ver qué líneas encajan (lo escribe en su salida estándar) como para obtener los nombres de los ficheros y las líneas. El comando sólo escribe el nombre del fichero 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 de globbing y de la shell. Para no tener que recordar cuales, la escaparemos siempre con comillas simples.

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 string. Además, en `/usr/share/dict` suele haber diccionarios con miles de palabras (una por línea) para hacer pruebas.

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
 $
```

2.21.4. 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 está diseñado para ser un filtro (de ahí lo de editor de flujos de texto). Es un editor: aplica el comando de `sed` a cada línea que lee y escribe el resultado por su salida. Sin el modificador `-n`, escribe todas las líneas después de procesarlas.

Si queremos utilizar expresiones regulares extendidas, hay que utilizar la opción `-E`.

Sed 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 [14].

Algunos comandos útiles de `sed`:

- `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. Es el que más se utiliza.

Ejemplos:

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

En el ejemplo de arriba `2` es una dirección. Una dirección es una forma que tiene `sed` de referirse a trozos de la entrada.

- `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.

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 primero 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/'
adios adios
adios adios
 $
```

El comando `sed -E 's/(`$e1$`)`e_2`(`e_3`).../\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.

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:

```
 $ echo 'hola adios
 mas menos' |sed -E 's/o./es/g'
heaa adiea
mas menes
 $ echo 'hola adios
 mas menos' |sed -E 's#o.#es#g'
heaa adiea
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 (menos recursos, menos paralelismo).

2.21.5. 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, 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[16] 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...). Al final imprime un salto de línea. No lleva paréntesis.
- `printf()` Función que imprime con formato. Lleva paréntesis. Ofrece control sobre el formato, de 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...). Por cada verbo tiene que haber otro parámetro extra cuyo contenido se sustituirá por el verbo en el formato que describe.

```
$  ls -l | awk '{ printf("Size: %08d KBytes\n", $5) }'
$
```

Variables especiales 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.

Por ejemplo, para imprimir la tercera y segunda columna de un fichero csv (el parámetro `-F` indica el separador que va a utilizar `awk` para las columnas):

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

```
 $ 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
 $
```

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, 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:

```
 $ 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 los ejemplos de arriba, se ven varios fragmentos del programa de `awk` entre llaves. Los trozos entre llaves se ejecutan una vez por línea, menos si tienen `BEGIN` o `END` antes de la llave abierta. En ese caso se ejecutan una vez antes y después, respectivamente, de procesar toda la entrada. Un programa en 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 -l | awk '/[Dd]esktop/{ print $0 }'
drwxr-xr-x  5 paurea paurea      4096 ene 17 14:03 Desktop
 $
```

Si quiero ser más preciso y que sólo una la novena columna encaje con la expresión regular, puedo escribir:

```
 $   ls -l | awk '$9 ~ /[Dd]esktop/{ print $0 }'
drwxr-xr-x  5 paurea paurea      4096 ene 17 14:03 Desktop
 $
```

El operador $\sim$ sirve para encajar expresiones regulares (evalúa a `true` si encaja y `false` en otro caso). Tiene su negación $!\sim$. Puedo utilizar otros operadores booleanos:

```
 $  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
 $
```

Ejemplo con variables definidas en el programa:

```
 $  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
 $
```

y pasadas como parámetro:

```
 $  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
 $
```

Dentro de las llaves, se puede escribir código más sofisiticado, `if`, `for`, etc. en una sintaxis similar a la de C. La variable reservada `next` pasa a la siguiente regla.

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 y `gsub` hace una similar a la de `sed` con `g` al final.

En el lenguaje `awk` hay arrays asociativos que son extremadamente potentes[17]. Por ejemplo, para imprimir cuantos procesos tiene ejecutando cada usuario en el sistema:

```
 $  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:

[17] Si has programado en `Python` son similares a sus diccionarios o a los mapas de `Go`.

```
 $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 númweo menor de argumentos que parámetros tiene definida y los 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 2.18: func.awk

```
1  #!/usr/bin/awk -f
2
3  function repname(name, repetitions, i) {
4      for(i = 0; i < repetitions; i++) {
5          printf("%d: %s\n", i, name);
6      }
7  }
8
9  {
10     repname($1, 3)
11 }
```

La función `repname` tiene dos parámetros y una variable local (`i`). Ál 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 y sólo hemos rascado la superficie. El lenguaje tiene multitud de funciones predefinidas, por ejemplo de matemáticas (por ejemplo, para números pseudoaleatorios, trigonometría) y cadenas de texto. Te recomendamos que leas detenidamente la página de manual `awk(1)`. Un buen libro para aprender más de `awk` es [14].

2.21.6. Visualización de ficheros

Hay dos comandos similares que se usan para visualizar ficheros de forma interactiva `less` y `more`. 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.

2.22. Árboles de ficheros

A veces es necesario recorrer un árbol entero de ficheros ejecutando comandos sobre cada uno de los ficheros contenidos en 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 significa algo (por ejemplo en `chmod` que significa *read* se suele utilizar la letra minúscula). 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`.

2.22.1. Du

El más sencillo y portable es `du`. El comando `du` (acrónimo de ***d**isk **u**sage*) 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 forma sencilla y portable de obtener un listado de ficheros y directorios.

```
 $ 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
.
 $
```

2.22.2. Ls recursivo

El comando `ls -aR` lista también 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
 $
```

```
 $ 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
 $
```

2.22.3. 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
$
```

```
$ 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
$
```

Find, sin embargo hace mucho más. Tiene dentro un motor de expresiones regulares y otro de globbing y permite ejecutar predicados lógicos sobre los ficheros para decidir qué comando o función ejecuta sobre ellos.

Ejemplos, 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, es sobre todo el path, no sobre el último componente como con globbing). Estamos utilizando `-regextype` para especificar las expresiones regulares extendidas de `egrep`.

```
 $ 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
 $
```

Después de un predicado se puede ejecutar un comando sobre cada fichero que encaje, uando 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
 $
```

Es bastante complicado de utilizar a la par que potente y es fácil escribir líneas de comando inexcrutable. Como cualquier gran poder, requiere una gran responsabilidad.

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

2.23. 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, 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(1)`.

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
 $
```

Y si quiero que ejecute en paralelo, puedo utilizar `-P` (ojo, la salida de `ls` la utilizamos 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
 $
```

2.24. Comandos varios

2.24.1. 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, ver, por ejemplo [15]. 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.

2.24.2. Cortar y pegar columnas

Hay dos comandos para cortar y pegar columnas, `cut(1)` y `paste(1)`. 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
 $
```

2.25. 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, en la entrega de una práctica.

Esto es lo que hace el comando `tar`. Su nombre, que significa alquitrán en inglés, viene 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). Es normal, después de meter un árbol de directorios en un fichero comprimirlo para que ocupe menos. Por eso, las herramientas que saben juntar ficheros, normalmente 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 excribir 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` tienen `.tar` como extensión. Si están comprimidos después pueden tener diferentes extensiones, por ejemplo, `.tar.gz` o `.tgz` si está comprimido con `gzip` `.tar.bz` o `.tbz` si está comprimido con `bzip2`.

Hay que ser cuidadoso y ejecutar tar desde el lugar adecuado para que al extraer no sobreescriba paths que no queremos. Si no está bien hecho el tar, se pueden quitar trozos del path, por ejemplo con `--strip-components`.

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

```
 $ 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
 $ tar -czvf prueba.tgz prueba
 $ 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
 $
```

Tar sabe llamar a otros comandos de compresión ver `tar(1)` (por ejemplo, la versión que tengo instalada 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`.

2.26. Limpiar rutas

Los *path* 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
 $
```

2.27. Creación de ficheros y directorios temporales

En algunos scripts es necesario guardar datos intermedios en ficheros y directorios temporales. El comando `mktemp(1)` 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.

```
 $ 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
 $
```

2.28. 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[18] 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
 $
```

Otro comando para visualizar ficheros binarios es `hexdump` que hace algo similar pero mostrando la salida en varios formatos al lado:

[18] Otra versión similar con más opciones es el comando `xxd`.

```
 $ 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
 $
```

Para ingeniería inversa más avanzada, ver el comando `radare`, `sudo apt install radare2` y su página de manual `r2(1)`. 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.

2.29. 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)[19].

```
 $ 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 a su entrada comandos. 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 escribimos con w.

Otro ejemplo:

[19] El comando ed es descendiente de qed. Ambos editores fueron escritos por Ken Thompson. Ambos eran editores que trabajaban con líneas. Todavía no había pantallas y se trabajaba con tiras de papel directamente que iba 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, la génesis de `grep`:

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

2.30. Ejercicios 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 2.19: 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` 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 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.

Esta 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 hubiese habido algún error. 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 arbol 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 con 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 parar en ese parámetro. 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 2.20: 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.

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 (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` (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.

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`:

Programa 2.21: 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 $dirs |sed -E 's/([^ ])/\\\1/g'| sed 's/ /|/g')
p=$(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[20]. 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 en los nombres de los ficheros. La expresión regular queda similar a `\/\t\m\p\/\a|\t\m\p\/\b`. Escapar todos los caracteres se 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 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`:

[20] Valorar compromisos, es decir en la solución de un problema multidimensional elegir a qué dimensiones se le dedican más recursos y a cuales 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.

```
 $ 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:
`man 1 file`

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 2.22: 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                 cp "$i" "$dirname"
35         fi
36 done
37
38 nfile=$(ls $dirname/|wc -l)
39
40 for i in $(seq -w 1 $nfile); do
41     fichname=$(ls $dirname/ | sed "$i"q|tail -1)
42     ext=$(echo "$fichname"|sed -E 's/.*(\.[^.]+)/\1/')
43     mv $dirname/$fichname $dirname/$i$ext
44 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. El comando `file` utiliza un conjunto de heurísticos (el número mágico de unix, un diccionario de condificaciones, valores e identificadores en varios offsets, el nombre del fichero...) 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. Las imágenes se identifican como `image/xxxxx` donde el lado derecho es el tipo de imagen, por ejemplo `image/gif`. 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). 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 mediante `sed` y `tail` (línea 41) y 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 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 2.23: 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 string `"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. Duespués 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 >>, 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 en otros ficheros. Los ficheros cuyo nombre comience con un carácter `x`, deben concatenarse en el 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 2.24: 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 2.23.

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*).

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 2.25: 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

- 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.
- Escribe un script que dado un fichero con paths te diga la profundidad máxima de los mismos. Pista: puedes contar `/`.
- 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.
- Escribe un script que haga exactamente lo mismo que `find . -ls -type f` utilizando `du` y `ls`.
- 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.
- 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`.
- 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.
- 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`
- 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`.
- 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.

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] N. Stephenson, *In the beginning... was the command line.* Avon Books New York, 1999.

[6] B. W. Kernighan and R. Pike, *Unix programming environment.* Prentice-Hall Software Series, 1984.

[7] S. Powers, J. Peek, T. O'Reilly, M. Loukides, and M. K. Loukides, *UNIX power tools.* .ºO'Reilly Media, Inc.", 2009.

[8] B. W. Kernighan and R. Pike, *The practice of programming.* Addison-Wesley Professional, 1999.

[9] C. Newham and B. Rosenblatt, *Learning the bash shell: Unix shell programming.* .ºO'Reilly Media, Inc.", 2005.

[10] 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.

[11] M. Lam, R. Sethi, J. Ullman, and A. Aho, "Compilers: Principles, techniques, and tools," 2006.

[12] M. Fitzgerald, *Introducing Regular Expressions: Unraveling Regular Expressions, Step-by-Step.* O'Reilly Media, 2012.

[13] J. E. Friedl, *Mastering regular expressions.* .ºO'Reilly Media, Inc.", 2006.

[14] A. D. Robbins and D. Dougherty, *Sed and Awk.* O'Reilly Media, Incorporated, 1997.

[15] C. J. Date, *An introduction to database systems.* Pearson, 8th ed., July 2003.

[16] N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME), Part One Format of Internet Message Bodies," RFC 2045, RFC Editor, November 1996.

Índice alfabético