Manual Linux eminentemente práctico, ZonaSiete.ORG

ZonaSiete.ORG Editors Team

Manual Linux eminentemente práctico, ZonaSiete.ORG:

por ZonaSiete.ORG Editors Team

El manual 'Linux eminentemente práctico' de ZonaSiete.ORG, es un pequeño pero práctico manual de referencia para usuarios medios de Linux. Su objetivo es plantear la solución a los problemas más frecuentes de la manera más sencilla posible, de tal forma que el usuario pueda resolver su problema cuanto antes para seguir trabajando. Asimismo, pretende servir de guía de aprendizaje progresiva para Linux para usuarios con conocimientos básicos de informática.

Este manual es desarrollado por el ZonaSiete.ORG Editors Team, compuesto por un grupo de usuarios con un dominio medio del sistema operativo Linux de habla hispana, que pretende que Linux gane adeptos y que además tengan un sitio donde resolver sus problemas fácilmente.

Siempre puede encontrar la última versión de este documento en http://www.zonasiete.org/manual.

Historial de revisiones

Revisión 0.6 8 de Enero de 2005
Añadidos capítulos 12, 13 y 14. Continua el glosario. Añadida información sobre Slackware. Re-revisados capítulos 1, 2, 3 y 4.
Revisión 0.5 5 de Febrero de 2004
Añadidos capítulos 10 y 11. Continua el glosario. Extendido capítulo 5
Revisión 0.4 11 de Agosto de 2003
Añadidos los capítulos 7, 8 y 9. Inicio del glosario.
Revisión 0.3 8 de junio de 2003
Añadidos los capítulos 4, 5 y 6
Revisión 0.2.2 31 diciembre 2002
Añadido el capitulo 3
Revisión 0.2.1 27 octubre 2002
Añadida guía general a la instalación
Revisión 0.2 28 agosto 2002
Comienzo de estructuración del documento

Tabla de contenidos

Lista de tablas

Capítulo 1. Introducción

En esta parte te ponemos al día sobre qué es GNU/Linux y algunos pequeños detalles que hay que conocer acerca de él antes de comenzar a instalarlo o usarlo.

Introducción a GNU/Linux, historia y cultura del Software Libre

Para empezar, GNU/Linux es un sistema operativo. Esto significa que es un conjunto de instrucciones que nos va a permitir "hacer cosas", cualquier tipo de tarea, con nuestro ordenador. Sus características más notables son:

- Proviene de UNIX, otro sistema operativo, y es casi un derivado de él.
- Es parte del proyecto GNU, lo que significa, entre otras muchas cosas, que es libre y que muchas veces no estás obligado a pagar por él. Puedes usarlo de modo gratuíto y además puedes modificar su código fuente, para adaptarlo a tus propias necesidades o para contribuir en su continuo desarrollo, en el que toman parte programadores de todo el mundo (tú puedes ser uno de ellos). Antes de seguir leyendo y aprendiendo sobre él, conviene que visites la página web del proyecto GNU (http://www.gnu.org) en la que se dan algunos detalles sobre qué es el proyecto, qué puedes hacer y qué no puedes hacer con el código fuente además de algunos otros detalles importantes.
- Es potente, seguro y estable. Debido a ello resulta un sistema operativo ideal para servidores ya que cumple muy bien esta función aunque como sistema operativo de oficina, de escritorio o de publicación también es excelente.
- Existen infinidad de programas para cualquier tipo de tarea que se quiera desarrollar y aunque muchos de ellos son gratuitos (no por ello de menos calidad que los comerciales, pues muchas veces los superan), también existen aplicaciones comerciales.
- Es un sistema multiusuario real y multitarea y funciona de manera muy productiva en redes. Desde sus raíces UNIX siempre lo ha sido. En definitiva Linux pone todo lo bueno de los grandes UNIX y algunas cosas más al alcance de todo el mundo. ¿Lo vas a dejar ahí? ¿Lo vas a desaprovechar? Un consejo, yo no lo haría.

Hablemos un poco de la historia de GNU/Linux. Comencemos presentando a quien podemos considerar como el creador de Linux. Su nombre es Linus Trovalds. Puede decirse que Linus Trovalds creó este sistema operativo en 1991. Escribió un KERNEL (parte principal de un sistema operativo) y lo dejó a disposición de sus amigos y luego de toda la Internet para que cualquiera pudiese mejorarlo. Siempre se ha seguido la regla POSIX para que el sistema operativo sea compatible con otros UNIX y de esta manera tener mayor software a su disposición, y que el que se cree para Linux valga también en otros UNIX. Algún tiempo más tarde, para adaptar el sistema operativo y facilitar la instalación, nacieron las compañías distribuidoras de Linux. Entre ellas destacan RedHat, Debian, Caldera, SuSE, Mandrake... Estas distribuidoras pueden cobrar por ofrecer sus productos en CD o en algún otro soporte, pero deben poner su código a disposición del público, si han utilizado software GNU (bajo licencia GPL, que dice entre otras cosas que si usas código GPL en tu programa, tu programa se convierte automáticamente en GPL). Actualmente Linus Torvalds sigue coordinando el trabajo de los programadores de todo el mundo en el núcleo del sistema operativo para mejorarlo y adaptarlo al nuevo hardware.

¿Cabe la posibilidad de preguntarse cuál es la mejor distribución de Linux? GNU/Linux tiene muchas distribuciones, y por lo general, no existe una distribución mejor que las demás, cada una de ellas tiene sus puntos a favor en determinados aspectos. Así, Debian es una distribución que no es realmente comercial en sí como lo son la mayoría de las otras. El desarrollo de Debian es muy parecido al voluntariado que existe en el desarrollo del kernel de Linux. Para los que se aproximan por primera vez a Linux, es cierto que Mandrake, si se tiene una máquina con recursos suficientes, ofrece una instalación y configuración sencillas y un entorno gráfico personalizado ideal para no perderse al principio. Claro que siempre depende de tí y si tienes la oportunidad de tratar con varias distribuciones, trabaja con la que te sientas más cómodo.

¿Puede haber problemas? Dependiendo del hardware que tengas en tu ordenador, porque Linux no soporta algunos dispositivos como Winmódems (módems diseñados para funcionar sólo con M$ Windows, en su mayoría internos), algunas tarjetas de sonido, etc.

¿Puedo tener Windows y Linux en la misma máquina? ¡Claro! Como veremos más adelante la clave está en particionar el disco duro, para que Windows use una parte y Linux use otra, de manera que no interfieran. También como veremos después, el cargador de arranque de Linux te permitirá elegir qué sistema operativo iniciar.

Esperamos que te hayas comenzado a interesar por GNU/Linux. Desde ZonaSiete, pretendemos prestarte toda la ayuda que podamos :-)

Sobre este manual

Este manual pretende ser ni más ni menos que una referencia para que cualquier usuario que tenga una duda pueda consultarlo y resolverla del modo más práctico posible gracias a las experiencias aportadas por otros usuarios. También está dirigido a las personas que quieren aprender Linux y hacerlo su sistema operativo de escritorio de cada día. En la sección de servidores, abordaremos configuraciones comunes para los programas y servicios más comunes.

Este manual NO pretende contradecir el contenido de la documentación oficial del proyecto The Linux Documentation Project (tldp.org (http://www.tldp.org/)), sino complementarlo a través de nuestras propias experiencias.

Todo el que quiera puede sugerir correcciones o escribir sobre alguna sección en particular. Tu ayuda a hacer el proyecto GNU/Linux cada vez más grande es desde ya agradecida :-)

Más detalles sobre los integrantes en Apéndice A.

Conceptos básicos GNU/Linux

En esta sección Vamos a tratar algunas cosas que tenemos que tener claras antes de instalar y comenzar a utilizar Linux. No tienes que aprenderlas de memoria (para eso está este manual), simplemente lee esto con atención y estarás en disposición de continuar aprendiendo sobre Linux.

El arranque de Linux

Veamos qué es lo que pasa cuando nuestra máquina arranca y tenemos que elegir un sistema operativo. Cuando la BIOS termina su chequeo, empieza a buscar dónde hay software para ser ejecutado. Si los CDROMs no son de arranque, pasa al primer disco duro por defecto. En el principio del disco duro o MBR (acuérdate, lo vas a ir más de dos veces con Linux) o en una partición queda instalado con Linux un cargador de arranque. Los dos más usados son LILO (LInux LOader) y GRUB. Nada más arrancar se ejecuta este cargador cuya misión permitir al usuario elegir sistema operativo de los que tenemos en nuestro ordenador quiere utilizar. Al instalarse, LILO (o GRUB) habrán sido configurados para poder arrancar cualquier sistema operativo que tengamos instalado, reconoce la mayoría de los existentes. Seleccionamos nuestra versión de Linux y el cargador da paso al Kernel de Linux que empieza a hacer sus chequeos y a montar (hacer utilizable) el sistema de archivos.

El sistema de archivos

El sistema de archivos es más o menos "la forma de escribir los datos en el disco duro". El sistema de archivos nativo de Linux es el EXT2. Ahora proliferan otros sistemas de archivos con journal-ising (si se arranca sin haber cerrado el sistema, no necesitan hacer un chequeo sino que recuperan automáticamente su último estado), los más conocidos son EXT3, ReiserFS y XFS.

La estructura de directorios que sigue Linux es parecida a la de cualquier UNIX. No tenemos una "unidad" para cada unidad física de disco o partición como en Windows, sino que todos los discos duros o de red se montan bajo un sistema de directorios en árbol, y algunos de esos directorios enlazan con estas unidades físicas de disco. MUY IMPORTANTE: Las barras en Linux al igual que en cualquier UNIX son inclinadas hacia la derecha, como se puede ver más abajo (ese es el motivo de que en internet sean inclinadas hacia la derecha ya que nació bajo UNIX y en Linux podremos aprovechar todas sus ventajas). Expliquemos esto más a fondo, incluyendo los directorios principales:

Tabla 1-1. Estructura de directorios en Linux

Directorio	Descripción
/	Es la raíz del sistema de directorios. Aquí se monta la partición principal Linux EXT.
/etc	Contiene los archivos de configuración de la mayoría de los programas.
/home	Contiene los archivos personales de los usuarios.
/bin	Contiene comandos básicos y muchos programas.
/dev	Contiene archivos simbólicos que representan partes del hardware, tales como discos duros, memoria...
/mnt	Contiene subdirectorios donde se montan (se enlaza con) otras particiones de disco duro, CDROMs, etc.
/tmp	Ficheros temporales o de recursos de programas.
/usr	Programas y librerías instalados con la distribución

Directorio	Descripción
/usr/local	Programas y librerías instalados por el administrador
/sbin	Comandos administrativos
/lib	Librerías varias y módulos ("trozos") del kernel
/var	Datos varios como archivos de log (registro de actividad) de programas, bases de datos, contenidos del servidor web, copias de seguridad...
/proc	Información temporal sobre los procesos del sistema (explicaremos esto más en profundidad posteriormente).

Nombres de archivos en Linux

Los nombres de archivos en Linux (como en todos los UNIX) distinguen mayúsculas de minúsculas, esto es, son "case sensitive". Los archivos README, readme, REadme y rEadme por ejemplo son archivos distintos y por lo tanto al ser nombres distintos pueden estar en el mismo directorio.

En Linux los archivos no tienen por qué tener una extensión. La suelen tener a modo orientativo, pero no es en absoluto necesario. Linux sabe qué contiene cada archivo independientemente de cuál sea su extensión. Por comodidad, podremos llamar a todos nuestros archivos de texto con la extensión .texto, o a todos nuestros documentos con la extensión .documento, de esta manera, podremos luego agruparlos más fácilmente.

Los ficheros y directorios ocultos en Linux comienzan su nombre por un punto (.)

Los nombres de archivos o directorios pueden ser muy largos, de más de 200 caracteres, lo cual nos da bastante flexibilidad para asociar el nombre de un archivo a lo que contiene. No obstante, hay ciertos caracteres que nunca se deberían utilizar a la hora de nombrar un archivo. Uno de ellos es el espacio, nunca llamaremos a un fichero con un nombre que contenga un espacio. Tampoco son recomendados otros caracteres raros como signos de puntuación (a excepción del punto), acentos o la ñ. En algunos casos Linux ni siquiera nos permitirá usarlos. Los recomendables son las letras A-Z, a-z, los números (0-9), el punto, el guión (-) y el guión bajo (_) para nombrar un archivo. Los acentos y la ñ tampoco se recomiendan.

El software adicional para GNU/Linux

Primero, hacer notar que LINUX NO ES WINDOWS. El software de Windows no funcionará en Linux, hay proyectos de emulación al respecto, que no recomendamos.

El software para Linux lo podemos encontrar de dos formas: en un paquete binario o en un paquete con su código fuente.

Los paquetes binarios

Existen varios tipos de paquetes binarios. Todos tienen una característica en común, y es que contienen código de máquina, no código fuente, por eso cada tipo de procesador necesita su propia versión

de cada paquete. Al haber varias distribuciones de Linux existen varios tipos de paquetes binarios, habiendo varias distribuciones que comparten sistema de paquetes. Los más comunes son:

- Los paquetes RPM: Los usan las distribuciones RedHat, Caldera, Madrake, SuSE y TurboLinux entre otras. Su uso está muy extendido y es posible instalar este tipo de paquetes mediante la aplicación rpm. El nombre de los paquetes rpm es del tipo nombredelpaquete_version_plataforma.rpm
- Los paquetes Debian (deb): Los usa la distribución Debian y sus derivadas. Es un sistema de paquetes muy potente y que facilita en gran medida la actualización del sistema, además de resolver las dependencias (qué paquetes necesitan a qué otros) y satisfacerlas instalando todos los paquetes necesarios automáticamente. La aplicaciones que gestionan este sistema de paquetes se llaman apt y dpkg. Los paquetes Debian se suelen nombrar de la forma nombredelpaquete_version_plataforma.deb
- Los paquetes tgz de Slackware los usa la distribución del mismo nombre, y siguen los principios de los dos anteriores, son paquetes binarios aunque tienen una estructura distinta.

Los paquetes de código fuente

Estos paquetes contienen los archivos que salen del ordenador del programador o programadores, lo que quiere decir que ya hay que aportar algo de nuestra parte para utilizar los programas que contienen. El proceso de instalación de este tipo de paquetes implica también una COMPILACIÓN, concepto que vas a escuchar más de tres veces. Una compilación nos permite que el programa que vamos a instalar se optimice totalmente para el tipo de componentes que tenemos en nuestro ordenador y el tipo de versión de GNU/Linux. Este programa que hemos compilado correrá más rápido que si nos hubiéramos limitado a instalar un paquete binario normal. La compilación de un programa requiere de unos COMPILADORES, que son unos programas que junto con unas LIBRERíAS de lenguajes de programación, consiguen transformar el código fuente en lenguaje de máquina. Trataremos más adelante cómo instalar software.

Son también de amplio uso los Source RPM, que son paquetes RPM pero que en vez de ser binarios, llevan código fuente. Mediante la instalación de este tipo de paquetes, lo que hacemos es crear un nuevo paquete optimizado (compilar un nuevo paquete) para nuestra máquina. Después instalamos este último.

Meta-Paquetes

Conocemos por meta-paquete a aquellos paquetes en los cuales no hay código fuente, o binarios, sino reglas sobre CÓMO construir e instalar dicho paquete. Estos paquetes los usan distribuciones como Gentoo Linux (http://www.gentoo.org), Linux From Scratch (http://www.linuxfromscratch.com) y algunos otros sistemas operativos como FreeBSD (http://www.freebsd.org). La ventaja que tienen estos paquetes es que son muy sencillos de generar, los binarios quedan optimizados para la máquina que los va a ejecutar, pero como todo, esto tiene una contrapartida.... las distribuciones que los usan pueden ser más complicadas de manejar y se necesitan conexiones muy rápidas y procesadores potentes si no se quiere estar mucho tiempo compilando los paquetes.

Los usuarios y los permisos de archivos

Linux es un sistema operativo multiusuario. Cada usuario generalmente tiene su carpeta de usuario en /home/usuario. Por defecto sólo puede escribir, modificar y borrar archivos dentro de esta carpeta. Ningún otro usuario (excepto root) puede acceder a los archivos que hay en este directorio, ni si quiera puede ver cuáles son. Este usuario -por defecto- puede leer en el resto de las carpetas que hay en el sistema de archivos excepto en la de root y las de otros usuarios. Todos los programas recuerdan las preferencias de cada usuario, e incluso un usuario puede instalar un programa sin que los otros usuarios tengan acceso a él (vale sí, root si tendrá, lo sé, lo sabemos :-); aunque instalando los usuarios tienen muchas limitaciones como veremos después. Un usuario no puede causar por este motivo daño al sistema ni cambiar su configuración de ninguna forma. ¿Y quién es ese tal root? En cualquier sistema UNIX, root es "el que todo lo puede". Es la excepción que confirma la regla, es el superusuario todopoderoso de estos sistemas. Cuando hagas login como root en una máquina GNU/Linux, siente el poder bajo tus teclas. Puedes hacer todo lo que se te pase por la cabeza. Pero ojo, tienes poder para lo bueno y para lo malo. Tienes acceso a todo el sistema, pero una equivocación... sería fatal. Puedes cargarte el sistema Linux y los preciados datos y configuraciones que tengas en él. Por esto, para tareas normales SIEMPRE entraremos al sistema como un usuario normal por los riesgos que se corren trabajando como root. Además NUNCA usaremos Internet como root. Incluso algunos programas no permiten ser ejecutados por root por motivos de seguridad. Como ya habrás adivinado, la contraseña de root se la guarda uno en la cabeza y se asegura de que no se le olvida, y por supuesto se preocupa uno de que nadie pueda acceder a ella en ningún fichero o de que no la ven cuando la escribimos. Si se cree que la han podido adivinar o están cerca, se cambia. Cuanto más larga, tediosa y sin sentido sea esta contraseña, más seguro estará nuestro sistema. Recuerda que tu máquina Linux es tan segura como segura sea tu contraseña de root.

¿Qué son los permisos? Todos y cada uno de los archivos y directorios del árbol jerárquico que monta nuestro sistema Linux tienen permisos. Estos permisos dicen, para cada usuario del sistema, si puede ejecutarlo, si puede ver su contenido o si puede borrarlo o modificarlo. Del mismo modo, cada elemento del sistema de archivos tiene un dueño. Por defecto, este dueño del elemento (tanto directorio como archivo) tiene acceso total a él y puede realizar todas las acciones posibles permitidas. El resto de usuarios pueden leer y ejecutar este elemento por defecto aunque todo esto se puede cambiar para cada uno de los elementos. Todos los archivos de las carpetas de sistema y configuración suelen tener a root como propietario. Los de la carpeta personal de cada usuario tienen a ese usuario como propietario, pero el resto de usuarios normales no tienen ningún permiso sobre estos elementos, y lo mismo ocurre con la carpeta de root (que se encuentra en la raíz, en /root).

El arranque de Linux II

Cuando el kernel ya ha montado el sistema de archivos, comienzan a inicializarse algunos procesos llamados "daemons" (demonios). Cada uno de estos demonios se inicia cuando el sistema arranca y durante su vida va acontrolar un proceso determinado, y va a permanecer en segundo plano (transparente para el usuario), no vamos a notar que se está ejecutando a menos que pidamos información a este proceso, lo detengamos o lo reiniciemos. Algunos de estos demonios son, por ejemplo, el servidor web, el servidor de correo, el cortafuegos, el servidor de nombres DNS... y muchos otros que gestionan varias tareas, en su mayoría servicios de servidor de red para servir de host a otras máquinas. Cuando todos esos demonios se hayan cargado, aparecerá ante nosotros una línea de texto de login (autentificación en un sistema) o bien una ventana de login gráfico. Vamos a empezar entrando como usuario normal primero.

Nombramiento de dispositivos y particiones

Debemos saber de qué manera nombra Linux a los discos duros que tenemos conectados a nuestra máquina y sobre todo a sus particiones. Todos los discos duros (IDE) comienzan su nombre como **hd**. Un ejemplo de nombre completo de disco duro sería `hda` y de la primera partición de ese disco duro sería `hda1`.

La 'a' significa que ese disco duro está conecatdo al IDE1 como maestro. Si fuera esclavo tendría la 'b', y si estuviera conectado al IDE2 como maestro, la 'c', y si estuviera al IDE2 como esclavo, la 'd'.

El número 1 indica que es la primera partición (primaria y no lógica) del disco duro en cuestión. La segunda geométricamente hablando (primaria) sería la 2 y así sucesivamente. La primera partición lógica de un disco duro se nombra con el número 5, independientemente de si pertenece a la primaria 1, 2, 3 ó 4. La segunda se nombraría con un 6 y así sucesivamente.

Con saber lo que significa hda1 o hdd2 o hdc5 es suficiente de momento.

Capítulo 2. Guía inicial para la instalación de una distribución

Ahora que ya nos hemos leído las secciones anteriores (y estamos un poco confusos si es la primera vez que tratamos con Linux), tenemos primeramente que relajarnos. La verdad es que, mientras estemos aprendiendo Linux, la primera vez que leamos cosas nuevas, nos van a resultar un poco confusas. La segunda vez que las leamos se nos clarificará un poquito la cosa, y si las practicamos un poco y después leemos por tercera vez la documentación de lo que hemos practicado, obtendremos una gran satisfacción interior :-) y habremos comprendido y aprendido algo más. Así que repetimos, nos tranquilizamos y nos preparamos para elegir la distribución de GNU/Linux que vamos a instalar en nuestro sistema.

Antes de instalar

Veamos qué necesitamos para instalar una distribución de Linux en nuestro PC.

Hardware

Como veremos más adelante en este manual, y se descubrirá cuando se tenga experiencia en el uso y administración de los sistemas Linux, es posible instalar Linux desde en un equipo "antiguo" hasta en los grandes servidores con varios microprocesadores y varios discos duros. Para instalar en un ordenador poco potente, necesitaríamos algunos conocimientos para personalizar a fondo la instalación que suponemos que no se tienen ahora. Así que vamos a buscar un equipo en el que una instalación normal funcione a una velocidad aceptable. Esto, lógicamente, va a depender de qué fecha sea la distribución que vayamos a instalar. Cuanto más reciente sea, necesitaremos un equipo más potente.

Las distribuciones actuales (quizás tengamos que ir cambiando esto ;-) pueden correr aceptablemente en un procesador a 450 MHz y 128 MB de memoria RAM. La memoria RAM va a ser un factor que incremente el rendimiento del sistema Linux en un gran porcentaje, mucho más de lo que pudiera parecernos en un principio.

En cuanto al espacio en disco, estamos en las mismas, dependerá de lo que vayamos a instalar, pero como empezarmos por una instalación estándar (no tocaremos mucho en un principio), pondremos un mínimo de 3GB de disco libres. Si tienes más espacio, puedes dejar 5 ó 6 GB los cuales te serán suficientes si tomas la inteligente decisión de seguir adelante con Linux :-)

Quedarnos con lo que ya tenemos en nuestro PC

Es común que cuando empezamos con Linux, si somos usuarios de Windows, se tenga un poco de miedo a lo desconocido y a pensar "¿podré seguir haciendo esto con Linux?" "¿Tardaré mucho en adaptarme?" y cosas por el estilo. Tanto es así, que querríamos conservar nuestra instalación de Windows al tiempo que instalamos Linux. Como comentamos en el Capítulo 1, eso es totalmente posible. Lo más fácil es dejar espacio libre en nuestra partición Windows, y con el programa ’Partition Magic’, redimensionar nuestra partición de Windows de tal modo que dejemos espacio libre en disco suficiente como para que en la instalación llenemos ese espacio con las particiones Linux. Claro, *’Partition Magic’* es un programa comercial, y un poco caro, así que si no se puede contar con él, todavía tenemos una esperanza más. Esta es *realmente tediosa* y consiste en usar un programa llamado *fips* que viene con algunas distribuciones de Linux, cuyo uso detallamos en las secciones siguientes de

guía inicial a la instalación. No obstante, la instalación de Mandrake también nos permite redimensionar nuestra partición Windows, incluye en la instalación un interfaz para gestionar las particiones de disco similar al de Partition Magic para Windows, así que si vamos a instalar Mandrake puede que estemos salvados.

No es muy común, pero raras veces se podrían perder los datos de nuestro disco duro. Evitemos males mayores, que no nos cuesta nada, y saquemos una copia de los documentos importantes o del trabajo que tenemos en nuestro sistema actual. El resto de software podríamos reponerlo desde los discos de instalación. Repetimos que esto no es nada frecuente pero *podría ocurrir*.

Elegir la distribución que instalaremos

Si, como suponemos puesto que estás leyendo esto, va a ser tu primera instalación de Linux, tendrás que elegir una distribución que no sea demasiado avanzada y te permita instalar sin grandes dificultades. Hoy esto no es dificíl, puesto que la mayoría de distros se instalan de manera muy sencilla. Podríamos elegir entre una de las siguientes:

- *Mandrake Linux*: Esta distribución de Linux es de las más amigables si nos estamos acercando a Linux la primera vez, calificada por muchos (y creemos que acertadamente) como la más sencilla para iniciarnos. La única "pega" (que no lo es realmente) de esta distribución es que instalará un montón de cosas que puede que no necesitemos, pero es el precio de una instalación sencilla. Es por esto por lo que deberíamos tener una máquina potente para instalar esta distribución sin que se nos vaya arrastrando. Ofrece un escritorio muy agradable y completo, además de varios asistentes sencillos para cambiar la configuración del hardware.

  La web oficial (http://www.mandrake.com/) de Mandrake nos permite descargarnos su distro. Deberemos buscar las imágenes ISO (archivos .iso) y descargarlas. En párrafos siguientes comentamos más acerca de las descargas. En la web oficial de Mandrake también puedes informarte de las características de esta distro.

- *RedHat/Fedora Linux*: Fedora es la distribución *libre* (gratuita) de la compañía RedHat, y RedHat Linux es su distribución comercial. Ambas son similares, sobre todo de cara al usuario medio. Podrían ser la segunda distribución más sencilla con la que tratar y por lo tanto igual de indicada para la instalación. Es algo más complicada que Mandrake para empezar, aunque la diferencia no es excesivamente notable. La web oficial de RedHat (http://www.redhat.com/) nos ofrece información sobre la distribución comercial y la web de Fedora (http://fedora.redhat.com/) nos informa y nos permite descargar la distribución gratuíta.

Si nos hemos decidido a descargar la distribución, nos aseguraremos de que tenemos una buena conexión :-) porque con un ancho de banda de bajada de 27 KBytes/s vamos a tardar unas 6 horas por CD. Si tenemos en cuenta que tanto Mandrake como RedHat son 3 CDs.... ¡¡18 horas!! ¡¡y eso con una buena conexión!!

Si tenemos la buena conexión, veamos qué es lo que tenemos que descargar. Buscaremos, los archivos *ISO (.iso)*, cada uno de ellos es un CD de los que luego grabremos. Así que sí, necesitamos tener una grabadora de CDs, unos cuantos CDs vírgenes y un programa de grabación funcionando en nuestro sistema operativo actual para grabar las imágenes de CDs ISO a los CDs vírgenes. En fin, como fuimos cocineros antes que frailes, en Nero para Windows es *Archivo -> Grabar Imagen*, esto es, no basta con grabarlo como si fuera un fichero normal, cada programa tiene su manera de grabar estas imágenes, así que buscaremos la del nuestro y las grabaremos

Si estáis en España, un buen mirror, rápido y actualizado, es *ftp.rediris.es*. Nos identificaremos como *anonymous* dando cualquier password, y nos moveremos por sus directorios hasta encontrar los archivos .iso de los discos de la distribución que queremos. Entonces los descargaremos con algún programa que nos permita continuar la descarga en caso de problemas. Cuando las tengamos descargadas las grabaremos.

Si no puedes descargar la distro desde Internet, puedes comprarla en caja. Las ventajas de comprarla son realmente muchas, y es que estarás seguro de que los discos no tendrán problemas de haberse descargado, tendrás más software que con las *download versions*, soporte por mail si tienes dudas en la instalación por parte de la compañía que crea la distro, y quizás lo mejor de todo, un montón de manuales impresos con los que guiarte y aprender a usar Linux paso a paso y tener una referencia impresa si la necesitas posteriormente. Si juntamos todo, el precio no es excesivo, ni mucho menos (compara con: Windows + Office + Photoshop + Soft de desarrollo + ..., sería muchísimo dinero). Precios aproximados; para Mandrake de 60 a 70 Euros (PowerPack) y para RedHat de 40 a 50 Euros (Personal Edition). Si vas a comprar la distro, otra que podrías evaluar sería SuSE Linux, que no ofrece la posibilidad de descargar los CDs de Internet, con un precio de 40 a 50 Euros (Personal). La decisión es tuya, estás ante tres buenas distribuciones que te servirán para empezar con Linux y para tu trabajo diario con multitud de herramientas de configuración y un escritorio agradable. Cualquiera de ellas sería una buena elección.

En el manual también vamos a tratar la distribución Debian GNU/Linux, es una distribución de gran calidad y que ofrece un gran rendimiento, con un sistema de paquetes para instalar software muy avanzado. No obstante, esta distribución puede ser demasiado difícil para iniciarnos en Linux, así que puedes empezar con una de las anteriores y posteriormente experimentar con alguna distribución avanzada como comentaremos después: Debian, Slackware o Gentoo.

Guias generales para la instalación

Instalar una distribución de Linux es realmente fácil en nuestros días. Los instaladores han avanzado mucho y hoy GNU/Linux se asemeja más bien a un Mac o un Windows que a un super-UNIX de alto nivel en lo que a la instalación se refiere (no ha perdido potencial por tener una instalación sencilla ;-)). Sería una pérdida de tiempo describir sección por sección la instalación de cada una de las distribuciones que recomendamos, así que lo que vamos a hacer es dar unos consejos generales que te resolverán las minimas dudas que puedan surgirte instalando. Puedes buscar los manuales de instalación en los discos o bien en las webs de las empresas distribuidoras si crees necesitar ayuda extra.

Redimensionado de particiones Windows con fips

Como ya comentamos, si queremos conservar nuestra partición Windows al instalar Linux, hay más bien poco que podamos hacer si no usamos 'Partition Magic' para hacer más pequeña nuestra partición Windows. Si vamos a instalar Mandrake Linux no tendremos problemas porque como hemos comentado antes, incluye un soft que nos permitirá hacer más pequeña nuestra partición Windows sin perder sus datos, para despues en el espacio libre crear las particiones Linux.

fips es una alternativa que pasamos a describir, aunque puede ser algo tediosa. Esta herramienta, aunque puede que la incluyan algunas distros más, es original de RedHat Linux.

> **Sugerencia:** Recuerda hacer una copia de seguridad de tu trabajo antes de proceder con este método

Este método está lejos de ser *eminentemente práctico*, además vamos a tener que depender de lo que Windows quiera hacer con nuestras particiones así que mejor será que recemos :-)

Procederemos siguiendo estos pasos:

> **Sugerencia:** Es posible que esto no funcione en algunas versiones de Windows

- En DOS (o una ventana de DOS) teclearemos: **`chkdsk`**
- Ejecutamos el scandisk en Windows (el corto será suficiente)
- Creamos un disquete de arranque con el programa FIPS incluido generalmente en el primer CD de RedHat Linux, escribiendo en DOS:

  `C:\` **`format a:/s`**

  A continuación le copiamos los archivos Restorbb.exe, Fips.exe y Errors.txt, localizados generalmente en el directorio `dosutils/fips20` del CD 1 de RedHat Linux.

- Desactivamos la memoria virtual de Windows.
- Cerramos todos los programas posibles y defragmentamos el disco duro para mover todos sus datos al principio.
- Reiniciamos el sistema y ponemos el disco de arranque. Si no aparece `C:\` escribimos algo como **`dir C:\`** y luego nos vamos a `A:\` donde escribiremos:

  `A:\` **`fips`**

  Con esto arranquermos `fips` por fin. Si no nos deja, probaremos a escribirlo desde `C:\`

- Con `fips` ejecutándose, primero elegimos el disco duro que queremos usar.
- Elegimos el número de partición que queremos dividir y le decimos que sí queremos comprobar la partición
- Le decimos que sí queremos hacer copia de seguridad del sector de inicio de nuestro disco duro, sin sacar el disquete le decimos que sí con la tecla Y. Se nos ofrecerá el espacio disponible para crear una nueva partición. Si es insuficiente o menor del que está en realidad significa que hay archivos ocultos que no han sido movidos al principio del disco duro con la defragmentación, o sea que tendremos que buscarlos y eliminarlos (y después defragmentar de nuevo, ya dijimos que esto no es cosa de coser y cantar...).
- Decidimos el tamaño nuevo de las particiones con las flechas izquierda y derecha, y cuando tengamos el que buscamos, presionaremos *intro*. A continuación podemos confirmar o cancelar la creación de particiones, continuaremos con la tecla C. Y a continuación con la tecla Y. Es todo; cuando finalice la tarea, el programa se cerrará y reiniciaremos sin quitar el disco. Luego escribiremos

  `fips -t`

  para comprobar que todo está correcto. Si no fuera así, FIPS te ayudará a dejarlo todo como estaba. Bueno, ahora quitamos el disco y reiniciamos, observando como Windows arranca normalmente.

Otros consejos útiles

Es posible que durante algún momento de la instalación te sientas perdido, así que aquí van una serie de consejos que te pueden ayudar.

Veamos algo sobre los sistemas de ficheros, las particiones y el cargador de inicio. Mandrake, por ejemplo, nos porporcionará la opción de redimensionar nuestras particiones Windows para crear espacio libre y añadir posteriormente las particiones para Linux. Para esto debemos coger la instalación personalizada (a veces llamada avanzada). Podemos ver más detalles en la documentación online de la distro en mandrake.com (http://www.mandrake.com). En el caso de RedHat deberíamos elegir la herramienta *DiskDruid* para crear las particiones; pero con ella ya deberemos tener espacio libre para crear las particiones Linux, puesto que esta herramienta no nos permite redimiensionar las particiones. En cuanto a los sistemas de ficheros (la manera de escribir los datos en el disco), elegiremos, si están disponibles, EXT3 o ReiserFS. Estos sistemas de ficheros no necesitan una comprobación si el PC se apaga inadecuadamente (también llamados de *journalising*). Crearemos tres particiones, una de mayor tamaño (al menos 3 GB) donde instalaremos el sistema, otra de tamaño medio dependiendo del disco que dispongamos (muy mínimamente 1 GB) donde se guardarán los documentos y configuraciones de los usuarios, y otra de unos 200 MB para SWAP (intercambio). Si fuera posible, la partición SWAP se creará en otro disco distinto a la partición principal. Cuando las tengamos creadas, las herramientas de particionado de los instaladores suelen tener un botón llamado *Modificar* o algo así. Ahí podremos escoger un tipo de sistema de ficheros, y elegir el *punto de montaje* . Para la partición principal escogeremos `/` como punto de montaje, para la de tamaño medio (la de los usuarios), elegiremos el punto de montaje `/home`, y la de SWAP le diremos que es del tipo SWAP y no se le asigna ningún punto de montaje. En lo que respecta al cargador de inicio, elegiremos GRUB si está disponible, y le diremos que queremos instalarlo en el primero disco duro de nuestro sistema (al principio de él), esto es, por ejemplo `/dev/hda`. Crearemos también un disco de inicio, lo etiquetaremos y lo guardaremos para arrancar Linux en caso de problemas después.

Otras preguntas en las que podrías dudar serían, por ejemplo, si quieres ejecutar las X al inicio. Para empezar con Linux, responde *sí* . El servidor X, es el servidor gráfico de Linux. KDE y GNOME son entornos de escritorio; instala, al menos, uno de ellos. Mozilla y Galeon son navegadores y Evolution y KMail son clientes de e-mail. Los instaladores nos dejarán probar resoluciones y profundidad de colores distintas para las X.

Deberías elegir una instalación personalizada, porque no es nada complicado aunque así suene. No instales servidores que no necesites (e igualmente con los servicios al inicio), puesto que harán que el sistema vaya más lento.

Ahora las distribuciones son fáciles de instalar, así que con estos consejos no deberías tener ningún problema :-) para instalar tu sistema Linux.

Las primeras veces que arranques Linux, antes de seguir leyendo, ¡investiga!, es una forma de aprender a ver cómo funciona el escritorio que vayas a usar.

Capítulo 3. Terminal o SHELL

Veamos primero lo que es una TERMINAL (o un SHELL, o un INTÉRPRETE DE COMANDOS, ambos casi sinónimos de TERMINAL). Probablemente muchos estéis acostumbrados a usar ENTORNOS GRÁFICOS (de ventanas) como los interfaces de Windows o de MacOS, y si habéis empezado a investigar el escritorio de Linux después de instalar vuestra distribución, habréis visto que hay varios escritorios para usar en Linux. Los entornos gráficos nos permiten ver colores, e interactuar con los botones de los programas para elegir opciones.

No obstante, la manera más habitual de administrar una máquina Linux (instalar paquetes, ver los registros de actividad, crear o modificar usuarios...) suele hacerse desde un terminal o intérprete de comandos, que es en modo texto, y generalmente nos muestra un PROMPT como ahora veremos. Un prompt es lo que el intérprete de comandos escribe automáticamente en cada lína antes de que nosotros podamos darle instrucciones mediante COMANDOS. Cada comando, generalmente , termina presionando la tecla INTRO para que éste sea recibido por la máquina y ejecutado.

Aparte de dar instrucciones a nuestra máquina, desde una terminal podremos editar textos, archivos de configuración, apagar y reiniciar el sistema, instalar nuevos programas, leer el correo, conectar al IRC, usar un navegador... y muchas cosas más. Aprenderemos que los sistemas gráficos no son imprescindibles, y lo que es mejor, al final te terminará gustando todo esto de los comandos (por mucho que ahora pienses que no). Para que te vayas haciendo una idea de cómo funciona todo esto y lo comprendas, vamos a pasar a explicar cómo funciona la terminal que usan por dfecto actualmente casi todos los sistemas Linux. En un capítulo posterior veremos detenidamente los sistemas gráficos en Linux.

Conceptos básicos de la terminal BASH

Cuando arranquemos nuestra máquina con Linux y termine de cargar todos los procesos iniciales, o bien veremos una pantalla gráfica de login o una de texto. Vamos a aprovechar este hecho para saber cómo podemos cambiar de terminal (podemos usar varias a la vez e ir cambiando entre ellas). Pulsando las teclas Ctrl+Alt+F1 accedemos a la primera terminal, pulsando Ctrl+Alt+F2 accedemos a la segunda, y así sucesivamente hasta Ctrl+Alt+F6 (Podremos volver al escritorio gráfico con Ctrl+Alt+F7 o a veces Ctrl+Alt+F5). Es posible que no podamos acceder a la primera terminal porque quizás esté siendo utilizada por los procesos de login gráfico si es que ya hemos ingresado desde allí. Las terminales se suelen nombrar tty0 para la primera, tty1 para la segunda, y así sucesivamente. También es posible, como descrubriremos luego, abrir una terminal en otra máquina desde la nuestra, esto es, manejarla remotamente.

Siempre que estemos ante una nueva terminal, veremos que aparece ante nosotros algo como `NOMBREDEMIMÁQUINA login:`. En cada terminal se ejecuta el programa que produce esta salida para que nos AUTENTIFIQUEMOS en el sistema (esto es, para que demostremos que somos usuarios registrados en el sistema y que podemos usarlo). El programa se llama `login`, y es de momento sólo importante que recuerdes que hay un programa que se ejecuta en cada terminal y que se encarga de autentificarnos en el sistema. Como vamos a practicar, *NO entraremos al sistema como root* . Bien, entremos al sistema en una de las terminales para practicar:

```
NOMBREDEMIMÁQUINA login: minombredeusuario
Password: mipassword
```

Supongamos que **`minombredeusuario`** es nuestro nombre de usuario (recuerda que Linux distingue mayúsculas y minúsculas para los usuarios y las contraseñas). Después de introducirlo pre-

sionaremos INTRO, y después introduciremos nuestra contraseña. Observa que no aparecen asteriscos mientras introduces tu contraseña (esto es por seguridad, para que nadie vea cuántos caracteres tiene tu password), pero puedes borrar y escribir normalmente. Cuando hayas terminado, presiona INTRO de nuevo.

Se comprobará si nuestro usuario y contraseña son correctos. De no ser así aparecerá algo como `Login incorrect. Please try again`. Esto significa que nuestro nombre de usuario o/y contraseña son incorrectos, así que lo intentaremos de nuevo. Cuando hagamos un login satisfactorio en el sistema, se nos mostrará algo como:

```
User [miusuario] logged in. Last login was in [FECHA] from [hostremoto_o_terminallocal]
```

Donde `[miusuario]` es mi nombre de usuario en la máquina, `[FECHA]` es la fecha del último ingreso al sistema y `[nombre_de_host_remoto_o_terminal_local]` es el nombre de la máquina desde donde ingresamos la última vez en la nuestra, o el nombre de la terminal local si el último login; por ejemplo `tty0`. En algunos casos se nos indicará las veces que se ha fallado antes de conseguir entrar con nuestro nombre de usuario. Como veremos después, podremos saber si alguien intenta hacerse pasar por nosotros con esta información.

Además (como después veremos en detalle), la máquina nos informará de si tenemos correo o correo nuevo en nuestro buzón local. Puede ser algo como `New mail for [misusuario]` o `Mail for [miusuario]`, dependiendo de si hay correo no leído o leído respectivamente. Si no lo hay o bien no se muestra nada o se muestra algo como `No mail for [miusuario]`.

Una vez dentro de nuestro sistema Linux, usando BASH (justo después de hacer login, pues es BASH la terminal por defecto), se nos mostrará lo siguiente en cada línea antes de que podamos introducir ningún comando:

```
[miusuario@nombredemimáquina directorioactual]$

miusuario@nombredemimáquina /directorio/actual$
```

A esto que se muestra antes de que podamos introducir un comando le llamamos PROMPT. Aparecerá cada vez que pulsemos INTRO. También puedes apreciar que hemos puesto dos líneas de prompt. La primera de ellas es la que verás si usas una distribución RedHat o basada en RedHat (Mandrake, SuSE...). La segunda es la línea que verás si usas una distribución Debian o basada en Debian. Analicemos las distintas partes del prompt más detenidamente:

- `miusuario` es el nombre de usuario con el que hemos entrado a la máquina.
- `nombredemimáquina` es el nombre de nuestra máquina sin el dominio (por ejemplo, una máquina llamada maquina23.midominio.com, aparecería aquí como maquina23).
- `directorioactual` es el nombre del directorio actual en el que estamos sin la ruta completa, sólo el directorio. Como nada más entrar al sistema, estaremos en `/home/miusuario`, lo más probable es que muestre solamente `miusuario`.

  `/directorio/actual` la RUTA COMPLETA al directorio actual (explicación más abajo), que es lo que mostrarán las distribuciones basadas en Debian. Nada más entrar deberían mostrar `/home/miusuario`, pero ahora veremos por qué esto no es así.

- $ indica que estamos en el sistema como un usuario normal, y no como `root`. Si entrásemos como root, en vez de un $ tendríamos un # en cada línea, lo cual debería recordarnos que "tecleemos con cautela" ;-)

Observaremos un ejemplo de lo que pasaría si entrásemos como root. Sería algo así:

```
[root@mimáquina root]#

root@mimáquina /root#
```

De nuevo, la segunda línea sería en distribuciones basadas en Debian. Recordamos que el DIRECTORIO PERSONAL (directorio donde cada usuario puede escribir y tiene sus archivos de configuración y su trabajo) de root es `/root`, mientras que para los usuarios normales es `/home/nombredeusuario`. Apreciamos que ahora la señal de "teclea con cautela" ;-) al final de la línea, #, somos root, tenemos la fuerza, pero debemos usarla para bien puesto que es nuestra máquina.

En cualquier momento, cerraremos nuestra sesión de terminal escribiendo `logout` o `exit` y presionaremos INTRO. El programa de login se lanzará automáticamente y podremos entrar de nuevo a la máquina como un usuario distinto o como root.

Las variables de entorno. La variable PATH

Lo primero que tienes que hacer es tranquilizarte, puesto que es realmente más sencillo de lo que su nombre aparenta. Estas variables se pueden llamar también VARIABLES DE SHELL. Y... ¿qué es un variable? *Una VARIABLE es un nombre al que le asignamos un valor.* Un ejemplo sería asignar al nombre "dias" el valor "7". Esto, en bash una vez iniciada la sesión podemos acerlo así:

```
[usuario@maquina usuario]$ dias=7

$  dias=7
```

A partir de ahora, para entrar comandos, lo representaremos de la segunda forma; presentaremos $ si los comandos han de ser ejecutados por un usuario normal o # si han de ser ejecutados por `root`, en lugar de mostrar en todos nuestros ejemplos todo el prompt completo. Debes recordar que el comando es lo que va detrás de $ o # según corresponda.

Sigamos. Lo que hemos hecho arriba ha sido ASIGNAR al nombre "dias" el valor de "7". Acabamos de crear una VARIABLE DE ENTORNO con nombre "dias" y valor "7". Sencillo, ¿no?. Podemos introducir texto como valor de una variable poniéndolo entre comillas. Para "referirnos" a las variables de entorno podemos usar el símbolo **$**. Para mostrar el contenido de una variable usaremos el COMANDO **echo**. Así, por ejemplo, para mostrar el valor de la variable `dias` que acabamos de crear, escribiríamos:

```
$ echo $dias
```

Lo cual nos daría como salida una nueva línea con el valor de `dias`, en este caso, 7. Nada más (por ahora) de todo esto, simplemente que nos vaya sonando.

Ahora debemos saber que BASH almacena algunas variables de entorno. Estas variables son necesarias, y contienen, por ejemplo, el tipo de sistema operativo que se está usando, la versión, el usuario activo, el directorio actual...

De todas ellas sólo nos interesa por ahora una llamada PATH (en mayúsculas, en Linux los nombres las distinguen, también en las variables). La variable PATH es muy importante. Puedes comprobar lo que contiene escribiendo:

```
$ echo $PATH
/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:/home/usuario/bin
```

Como podemos observar, lo que contiene esta variable son rutas de directorios separadas por dos puntos. `/bin`, `/usr/bin`, y `/usr/local/bin` son ejemplos de rutas de directorios. ¿Para qué sirven? Es sencillo. Cuando introducimos un comando en BASH (como los que ahora vamos a ver, y como ya hemos visto con `echo`), se busca por el archivo del mismo nombre que ejecuta ese comando, en todos los directorios que la variable `$PATH` tiene asignados. Si encuentra el archivo (que se llama igual que el comando que tecleemos y contiene unas instrucciones) lo ejecutará. Si no lo encuentra, nos mostrará un mensaje de error. Todo esto puede parecer complicado, pero verás que no lo es en absoluto.

Notemos que si hacemos login como root y mostramos el contenido de `PATH`, ésta contiene algunas cosas más, como `/usr/sbin` y `/sbin` por ejemplo. Estos directorios sólo los tendremos si entramos desde la pantalla de login como root.

Pero podemos añadir más directorios a nuestra variable `PATH`. Bien porque trabajemos mucho con ellos o porque es necesario para la ejecución de algún programa. La forma de hacerlo es la siguiente:

```
$ PATH="$PATH:/home/usuario/programas"
```

¿Recordamos que la primera `$` sólo indica que estamos trabajando como un usuario normal? Entonces perfecto. Lo que hace esta línea es simple. Al contenido de PATH añade unos dos puntos y a continuación añade el directorio que nosotros queramos a nuestra PATH. El resultado es que si mostramos haora el valor de PATH con **`echo`** obtendremos todo lo que teníamos anteriormente más el directorio que hemos añadido.

Para aspectos más avanzados sobre las variables de entorno y la programación en shell se pueden varios manuales o guías enlos sitios de documentación oficial de Linux: The Linux Documentation Project (http://www.tldp.org/) o TLDP-es (LuCAS) (http://lucas.hispalinux.es/). De todos modos trataremos la shell un poco más a fondo en secciones posteriores.

Comandos básicos en la terminal BASH

Ciertamente, todos estos conocimientos que estamos citando pueden casi siempre aplicarse a otras terminales como `sh`, pero nos referimos a BASH por ser la que viene por defecto con Linux. Ahora vamos a explicar el concepto de DIRECTORIO DE TRABAJO o directorio actual. En esta línea:

```
[usuario@maquina usuario]$
```

estamos trabajando en el directorio `/home/usuario`. ¿Qué quiere decir esto? Pues es fácil, que podemos "hacer referencia" a un archivo desde un comando simplemente con el nombre de archivo y no su ruta completa si éste está en `/home/usuario` porque es el directorio de trabajo actual. Un ejemplo rápido, vamos a editar un archivo para comprender esto, no te preocupes por el comando `vim`, puesto que lo veremos más adelante. Si estamos en `/home/usuario` y tenemos un archivo que se llama `/home/usuario/mitexto`, para editarlo, bastaría con hacer:

```
[usuario@maquina usuario]$ vim mitexto
```

Como estamos trabajando en el directorio `/home/usuario` y el archivo que queríamos editar también está en el directorio de trabajo, el editorlo encontraría sin problemas. Esto sería equivalente a hacer:

```
$ vim /home/usuario/mitexto
```

pero como acabas de ver, teclear la ruta completa no es en absoluto necesario si el `objetivo` de nuestro comando está en el directorio de trabajo.

> **Sugerencia:** Puedes salir del editor pulsando: `Esc :wq INTRO`
>
> Para más información sobre VIM mira el Capítulo 4.

Referencias relativas

Este concepto es también sencillo. Partimos de que estamos otra vez en `/home/usuario` (de nuevo no te preocupes por `vim`, el comando del editor, lo usamos como un ejemplo para que veas las referencias, no es necesario que pruebes esto porque vamos a explicarlo más adelante), y queremos editar un archivo que está en `/home/usuario/documentos/mitexto`. Al estar en `/home/usuario`, bastaría con hacer:

```
$ vim documentos/mitexto
```

Lo cual sería equivalente a hacer:

```
$ vim /home/usuario/documentos/mitexto
```

Pero como ya hemos visto, con la referencia relativa al directorio en el que nos encontramos es suficiente. Por esto ya has podido observar que podemos referirnos a los archivos de dos formas distintas: mediante una *referencia relativa al directorio actual* en el que nos encontramos o mediante una REFERENCIA ABSOLUTA. Las referencias absolutas comienzan siempre desde la raíz del sistema de archivos, `/`, y por eso siempre empiezan con `/` e indican la RUTA COMPLETA al archivo al que nos estamos refiriendo.

El comando `cd`

Este sencillo comando, lo que nos va a permitir es cambiar nuestro directorio de trabajo (o directorio actual). Es una abreviatura (como muchos otros comandos) de lo que hace, en este caso de Change Directory. Podemos necesitar cambiar nuestro directorio de trabajo si vamos a trabajar con muchos archivos que estén en el direcotrio al que nos cambiamos, y no queremos teclear siempres su ruta completa por resultar esto demasiado incómodo y largo. Partimos de `/home/usuario`, y nos queremos cambiar a `/usr/local`, que es donde residen muchas cosas personalizadas de nuestro sistema. Se podría hacer de la siguiente forma:

```
[usuario@maquina usuario]$ cd /usr/local/
```

Y entonces ya estaríamos en `/usr/local`:

```
[usuario@maquina local]$
```

Si simplemente escribimos `cd`, sin ningún argumento (dato que necesita un comando para operar de una manera u otra, con un archivo u otro... como después veremos), lo que hará será llevarnos a nuestro directorio personal:

```
[usuario@maquina local]$ cd
[usuario@maquina usuario]$
```

Como puedes ver, ahora nuestro directorio de trabajo es `/home/usuario`, que es nuestro directorio personal.

Pero hay algunas cosas que no hemos contado de las referencias relativas. Es que dos puntos seguidos **`..`** se refieren al directorio inmediatamente anterior (superior) al que nos encontramos, y un punto **`.`** se refiere al directorio en el que nos encontramos (directorio actual). Si estando en `/home/usuario` escribes **`cd .`**, te darás cuenta de que no cambia de directorio, se queda en el actual, de la misma manera que si hubieras hecho **`cd ..`** te habría llevado a `/home` , el directorio inmediatamente superior. Estos dos comandos equivaldrían a **`cd ./`** y **`cd ../`**; es lo mismo, pero es realente con la barra como deben utilizarse estos comandos (puesto que nos estamos refiriendo a directorios). Utilizarlos sin ella, puede dar lugar a confusión y equivocaciones (recuerda que los archivos y directorios ocultos comienzan su nombre por `.`). Ahora ya sabes que para poder ir a `/home/usuario/documentos` podríamos haber hecho:

```
$  cd ./documentos
```

Lo cual no es muy eficiente en este caso, pero como veremos más adelante, es de obligada utilización para ejecutar un comando en el directorio actual si este directorio actual no está en la PATH. Por ejemplo para ejecutar un programa (como veremos más adelante) que estuviese en `/home/usuario`, al no estar esta ruta en `$PATH`, necesitaríamos hacerlo así (estando en `/home/usuario`):

```
$ ./miprograma
```

Te habrás dado cuenta de que con una referencia relativa es más cómodo, más sencillo y más rápido. Si simplemente dijésemos desde `/home/usuario` **`miprograma`** a BASH, no lo encontraría, y nos devolvería que no ha encontrado el comando. ¿Por qué? Pues es fácil, `/home/usuario`, que es donde reside `miprograma`, no está en `$PATH`. Bueno, la ejecución para después, ahora sigamos:

Desde `/home/usuario` podríamos haber ido a `/usr/local` con una referencia relativa también:

```
[usuario@maquina usuario]$ cd ../../usr/local
```

Este comando sube dos niveles (al directorio inmediatamente superior a `/home/usuario`, que es `/home`, y luego al directorio inmediatamente superior a `/home` que es `/`; una ez allí busca mediate referencia relativa a `/` el directorio `usr`, y una vez dentro, mediante una referencia relativa al directorio `usr`, busca el directorio `local`. En este caso es más rápido introducir **`cd /usr/local`** que usar una referencia relativa al directorio actual. Generalmente, usaremos referencias absolutas (al sistema de archivos, `/`) cuando lo que queremos buscar (en este caso un directorio) se encuentre varios niveles por encima del directorio actual, o en una ruta distinta como en este ejemplo.

Archivos y directorios

Ahora vamos a trabajar con archivos y directorios, vamos a crear directorios, los borraremos, listaremos archivos... Todo esto no es difícil de hacer. Si recordamos, es preferible que trabajemos como un usuario normal y no como root, para no dañar nuestro sistema por accidente intentando borrar un directorio importante. Empezaremos con obtener un listado de los contenidos del directorio `/`. Este se hace con el comando **`ls`**. Las opciones (indican al comando la forma en la que debe trabajar) que tiene el comando **`ls`** son principalmente **`-a`** y **`-l`**. **`-a`** indica que se muestren todos los archivos, incluso los ocultos, y **`-l`** muestra un listado largo, con detalles como el usuario al que pertenece el archivo, los permisos de ese archivo y algunas cosas más. Los parámetros que toma **`ls`** principalmente es un directorio. Si no se especifica ninguno, saca un listado del directorio actual. Veamos algunos ejemplos:

```
[usuario@maquina usuario]$ ls
Desktop Mail documentos kernelconfig programas proyectos
[usuario@maquina usuario]$
```

Esto es un listado de los archivos y directorios que contiene el directorio actual, en este caso `/home/usuario`. En algunas distribuciones, los directorios aparecen en color azul, los ejecutables en color verde, etc. pero muchas veces no hay distinción.

```
[usuario@maquina usuario]$ ls -l

total 44
drwx--S--- 3 usuario usuario 4096  Mar 28 10:51 Desktop
drwx--S--- 2 usuario usuario 4096  Mar 28 01:40 Mail
drwxr-sr-x 3 usuario usuario 4096  Mar 28 10:24 documentos
-rw-r--r-- 1 usuario usuario 18417 Mar 27 21:31 kernelconfig
drwxr-sr-x 4 usuario usuario 4096  Mar 28 10:28 programas
drwxr-sr-x 7 usuario usuario 4096  Mar 28 00:18 proyectos
```

Como puedes ver, la opción **`-l`** nos devuelve un listado largo, en el que apreciamos permisos (que explicaremos más adelante), dueño y grupo de archivo, fecha, nombre, tamaño, y algunos otros detalles.

```
[usuario@maquina usuario]$ ls -a

.   .bash_logout  .mcoprc  Mail
..    .bash_profile  .mozilla  ccmsn
.DCOPserver_usuario  .bashrc  .ssh   documentos
.DCOPserver_usuario_:0  .viminfo  kernelconfig
.ICEauthority   .gtkrc-kde  .wmrc   programas
.MCOP-random-seed  .kde   .xchat   proyectos
.Xauthority   .kderc   .xsession-errors
.bash_history   .mcop   Desktop
```

Aquí puedes ver un listado de todos los archivos y directorios, incluidos los ocultos. Podríamos emplear las dos opciones juntas, como **`ls -la`** o **`ls -al`**, lo que nos devolvería un listado largo con todos los archivos, ocultos incluidos. Si tienes cualquier duda o quieres saber qué parámetros acepta un comando, simplemente escribee **`comando --help`** y verás una lista de opciones y argumentos que puedes usar con ese comando. En el caso de `ls`:

```
$ ls --help

Modo de empleo: ls [OPCIÓN]... [FICHERO]...
Muestra información acerca de los FICHEROs (del directorio actual por defecto).
Ordena las entradas alfabéticamente si no se especifica ninguna de las
opciones -cftuSUX ni --sort.

Los argumentos obligatorios para las opciones largas son también obligatorios
para las opciones cortas.
  -a, --all                  no oculta las entradas que comienzan con .
  -A, --almost-all           no muestra las entradas . y .. implícitas
      --author               imprime el autor de cada fichero
  -b, --escape               imprime escapes octales para los caracteres no
                             gráficos
      --block-size=TAMAÑO    utiliza bloques de TAMAÑO bytes
  -B, --ignore-backups       no muestra la entradas que terminan con ~
  -c                         con -lt: ordena por ctime y muestra ctime (fecha
                               de última modificación del fichero)
                               con -l: muestra ctime y ordena por nombre
                               en cualquier otro caso: ordena por ctime
  -C                         muestra las entradas por columnas
      --color[=CUÁNDO]       especifica si se usará color para distinguir los
                               tipos de ficheros. CUÁNDO puede ser 'never',
                               'always' o 'auto'
  -d, --directory            muestra las entradas de los directorios en lugar
                             de sus contenidos
  -D, --dired                genera el resultado para el modo 'dired' de Emacs
  -f                         no ordena, utiliza -aU, no utiliza -lst
  -F, --classify             añade un indicador (uno de */=@|) a las entradas
      --format=PALABRA       across -x, commas -m, horizontal -x, long -l,
                               single-column -1, verbose -l, vertical -C
      --full-time            como -l --time-style=full-iso
  -g                         como -l, pero no muestra el propietario
  -G, --no-group             no muestra la información del grupo
  -h, --human-readable       muestra los tamaños de forma legible
                             (p.e. 1K 234M 2G)
      --si                   análogo, pero utilizando potencias de 1000,
                             no de 1024
  -H, --dereference-command-line  sigue los enlaces simbólicos en la línea de
                                  órdenes
      --indicator-style=PALABRA  añade un indicador con estilo PALABRA a los
                                 nombres de las entradas: none (predeterminado),
                                 classify (-F), file-type (-p)
  -i, --inode                muestra el número de nodo-i de cada fichero
  -I, --ignore=PATRÓN        no lista las entradas que coincidan (encajen)
                             con PATRÓN de shell
  -k                         como --block-size=1K
  -l                         utiliza un formato de listado largo
  -L, --dereference          al mostrar la información de un fichero para un
                               enlace simbólico, muestra la información del
                               fichero al que apunta el enlace en lugar de la
                               del propio enlace
  -m                         rellena el ancho con una lista de entradas
                             separadas por comas
  -n, --numeric-uid-gid      como -l, pero muestra los UIDs y GIDs numéricos
  -N, --literal              muestra los nombres literalmente (no trata p.ej.
                             los caracteres de control de forma especial)
```

```
  -o                         como -l, pero no muestra el grupo
  -p  --file-type            añade un indicador (uno de /=@|) a las entradas
  -q, --hide-control-chars   imprime ? en lugar de los caracteres no gráficos
      --show-control-chars   muestra los caracteres no gráficos tal y como
                             son (predeterminado a menos que el programa sea
                             'ls' y la salida sea un terminal)
  -Q, --quote-name           encierra los nombres de las entradas entre
                             comillas
      --quoting-style=PALABRA  utiliza el estilo de cita PALABRA para los
                               nombres de las entradas:
                               literal, locale, shell, shell-always, c, escape
  -r, --reverse              invierte el orden, en su caso
  -R, --recursive            muestra los subdirectorios recursivamente
  -s, --size                 muestra el tamaño de cada fichero, en bloques
  -S                         ordena los ficheros por tamaño
      --sort=PALABRA         extension -X, none -U, size -S, time -t, version -v
                             status -c, time -t, atime -u, access -u, use -u
      --time=PALABRA         muestra la fecha según PALABRA, en lugar de la
                             fecha de modificación:
                               atime, access, use, ctime ó status; utiliza
                               la fecha especificada como clave de ordenación
                               si --sort=time
      --time-style=ESTILO    muestra la fecha utilizando el estilo ESTILO:
                               full-iso, long-iso, iso, locale, +FORMATO
                             FORMATO se interpreta como en 'date'; si FORMATO
        es FORMATO1<nuevalínea>FORMATO2, FORMATO1 se
                             aplica a los ficheros no recientes y FORMATO2
                             a los ficheros recientes; si ESTILO está precedido
                             por 'posix-', ESTILO surte efecto solamente fuera
                             del local POSIX
  -t                         ordena por la fecha de modificación
  -T, --tabsize=COLS         establece los topes de tabulación a cada COLS
                             en lugar de 8
  -u                         con -lt: ordena por atime y muestra atime (fecha
                               de último acceso al fichero)
                               con -l: muestra atime y ordena por nombre
                               en cualquier otro caso: ordena por atime
  -U                         no ordena; muestra las entradas en el orden del
                             directorio
  -v                         ordena por versión
  -w, --width=COLS           establece el ancho de la pantalla en lugar del
                             valor actual
  -x                         muestra las entradas por líneas en vez de por
                             columnas
  -X                         ordena alfabéticamente por la extensión de la
                             entrada
  -1                         muestra un fichero por cada línea
      --help     muestra esta ayuda y finaliza
      --version  informa de la versión y finaliza

TAMAÑO puede ser (o puede ser un entero seguido opcionalmente por) uno
de los siguientes: kB 1.000, K 1.024, MB 1.000.000, M 1.048.576, y así
en adelante para G, T, P, E, Z, Y.

Por defecto, no se emplea color para distinguir los tipos de ficheros. Esto
equivale a usar --color=none. Usar la opción --color sin el argumento opcional
CUÁNDO equivale a usar --color=always. Con --color=auto, sólo se muestran
```

```
los códigos de color si la salida estándar está conectada a un terminal (tty).

Comunicar bugs a <bug-coreutils@gnu.org>.
```

(La ayuda puede no estar en castellano). Como puedes ver, la cantidad de opciones es impresionante. Te hemos citado las que de momento te pueden ser necesarias. Es obvio que no es necesario que recuerdes ninguna excepto las dos vistas, cuando el resto puedes obtenerlas mediante `--help`. Recuerda que puedes hacer esto con casi cualquier comando.

Podríamos listar los contenidos de otro directorio sin cambiarnos a él:

```
$ ls /usr/local/
bin include lib man sbin share src
```

Pasemos a ver ahora cómo podemos crear un directorio. Como pienso que habrás intuido, no puedes crear un directorio en un lugar en el que no tienes permiso para ello. Generalmente, un usuario normal sólo tendrá permiso para ello en su directorio de usuario, `/home/usuario`, así que cámbiate ahí si no es donde estás. Para crear directorios podemos utilizar tanto referencias relativas al directorio actual como referencias absolutas al sistema de archivos. Para crear directorios, basta con usar el comando **`mkdir`**. Crearemos un directorio en `/home/usuario` llamado pruebas, y otro dentro de este llamado `pruebas1` como ejemplo:

```
$ cd /home/usuario
```

Primera forma de hacerlo:

```
$ mkdir pruebas
$ cd pruebas
$ mkdir pruebas1
```

Segunda forma de hacerlo:

```
$ mkdir pruebas
$ mkdir pruebas/pruebas1
```

Tercera forma de hacerlo:

```
$ mkdir /home/usuario/pruebas
$ mkdir /home/usuario/pruebas/pruebas1
```

Creo que ya lo has entendido, ¿verdad? Puedes moverte entre esos directorios con el comando **`cd`** como ya explicamos antes, así practicas un poco. Ahora vamos a mover y copiar archivos a esos directorios. Para mover archivos y directorios usaremos el comando **`mv`**. Date cuenta que este comando también nos permite renombrar un archivo. Supongamos que tenemos un archivo llamado `miarchivo` en `/home/usuario`. Vamos a renombrarlo a `minuevoarchivo` (es recomendable no usar espacios en los nombres de archivo, puedes usar guiones bajos (_) en sustitución de los espacios; asimismo es recomendable como ya tratabamos, no usar acentos en los nombres de archivo):

```
$ cd /home/usuario
```

Primera forma de hacerlo:

```
$ mv miarchivo minuevoarchivo
```

Segunda forma de hacerlo:

```
$ mv miarchivo /home/usuario/minuevoarchivo
```

Tercera forma de hacerlo:

```
$ mv /home/usuario/miarchivo minuevoarchivo
```

... y así algunas formas más, pero creo que ya lo has entendido, aquí también podemos emplear referencias relativas al directorio actual y referencias absolutas al sistema de archivos. Podemos renombrar directorios también, y moverlos unos dentro de otros. Para mover un directorio llamado `programas` y llevarlo dentro de otro llamado `pruebas`, haríamos algo como esto:

```
$ mv programas pruebas/
```

De esta forma el directorio `programas` se queda dentro de `pruebas`. Como ves, el comando **`mv`** toma como primer argumento el fichero o directorio origen, y como segundo argumento el destino; recuérdalo. Podríamos reestablecer el directorio `programas` al nivel anterior donde estaba:

```
$ cd pruebas/
$ mv programas ../
```

Como habrás podido apreciar y ya hemos dicho, el comando **`mv`**toma primero lo que se quiere mover y después a dónde se quiere mover. Para dominarlo totalmente, lo mejor es que practiques, es la única forma de comprender cómo se comporta bajo determinadas situaciones este comando, y de que tú lo domines.

Para borrar un directorio, si este esta vacío, usaremos el comando **`rmdir`**. Por ejemplo vamos a borrar los directorios `pruebas1` y `pruebas` que creamos anteriormente. Si recuerdas, `pruebas1` estaba dentro de `pruebas`:

```
$ ls
pruebas
$ rmdir pruebas
rmdir: ‘pruebas’: El directorio no está vacío / Directory not empty
# Como ves, no podemos borrarlo porque no está vacío

$ cd pruebas/
$ ls
pruebas1
$ rmdir pruebas1
# pruebas1 está vacío y por lo tanto se borra

$ cd ../
$ rmdir pruebas
# Ahora pruebas está vacío y también se borra
```

(Las líneas que comienzan por # son explicaciones del manual para que comprendas lo que pasa). Está claro que hacer todo esto para borrar un directorio y todo lo que éste contenga no es del todo cómodo. Unas líneas más adelante veremos cómo es posible borrar todo un directorio con todo lo que contenga por debajo de él (siempre que tengamos los permisos adecuados sobre todo lo que contiene y sobre el directorio mismo).

El comando **`rm`** nos permite borrar uno o varios archivos. Por ejemplo, borraremos el archivo `minuevoarchivo`:

```
$ rm minuevoarchivo
```

Si tenemos los permisos adecuados sobre él, `minuevoarchivo` será borrado, en la mayoría de los casos sin preguntarnos, por lo cual debemos tener bastante cuidado con lo que hacemos con los archivos de nuestro sistema, porque una vez borrados es imposible volver a recuperarlos (¿para que los borraríamos entonces si siguiesen estando ahí?). Al igual que todos los comandos anteriores, **`rm`** es muy potente y sobre todo muy flexible. Veremos cómo es capaz de borrar un directorio con todo lo que tenga por debajo.

Para esto se usa, principalmente, la opción **`-r`**. Esta opción le dice a **`rm`** (y a algunos otros comandos) que opere con recursividad, esto es, que se vaya adentrando nivel por nivel debajo del directorio que le hemos indicado y que vaya borrando todo lo que encuentre a su paso. De este modo, el directorio `pruebas/` que teníamos antes, podríamos haberlo borrado así:

```
$ rm -rf pruebas/
```

La opción **`-r`** como hemos dicho irá borrando todo, terminando por borrar el directorio, y la opción **`-f`** es para que no nos pregunte si queremos que borre o no cada subdirectorio del que nosotros le indicamos. Recuerdo que el caracter **`*`** hace referencia a todos los archivos de un directorio por lo que si quisieramos haber borrado todo lo que tuviese dentro el directorio `pruebas/` pero sin borrar éste, podríamos haber hecho:

```
$ rm -rf pruebas/*
```

La opción **`-r`** nos permite de igual forma copiar un directorio entero. Pogamos que queríamos copiar el directorio `pruebas` (con todo su contenido) a un nuevo directorio llamado `pruebas2` que crearemos al mismo nivel que `pruebas`:

```
$ cp -rf pruebas/ pruebas2/
```

Más o menos ya tienes una idea de cómo se maneja el sistema de archivos desde la consola. Para entender y dominar todo esto no hay más que practicarlo. Si quieres saber cómo crear archivos, la sección del editor **`vim`** puede servirte, ahora vamos a comentar algunos detalles más para referirnos a archivos en BASH.

Piensa que tenemos un directorio que se llama `mis libros`. Como puedes observar, este nombre de directorio tiene un espacio, lo cual no es nada recomendable. La manera de "nombrarlo" desde la terminal es la siguiente: **`mis\ libros`**. El espacio se sustituye por **`\`** (barra invertida y un espacio).

Piensa ahora que tenemos que referirnos a un archivo que tiene cinco letras pero que desconocemos cuál es la última (o cualquiera) de estas letras. Por ejemplo, un archivo que se llame `maybe`, nosotros desconocemos la letra "b". Podríamos referirnos a él usando **`may?e`**, y la terminal lo encontraría inmediatamente.

El asterisco ***** puede referirse a todos los archivos de un directorio si lo usamos solo. Por ejemplo, borramos todos los archivos de un directorio:

```
$ rm *
```

O bien puede referirse a archivos que contengan una cadena de caracteres o que empiecen o terminen por una determinada cadena de caracteres. Por ejemplo todos los archivos ocultos (que empiezan por "."):

```
$ rm .*
```

Otro ejemplo, borramos todos los archivos que contengan la cadena "so" en su nombre:

```
$ rm *so*
```

Todo esto es muy sencillo. Vamos a terminar con una función muy importante de BASH, que es completar el nombre de archivo o directorio por nosotros. Supongamos que queremos entrar en un directorio dentro del actual que se llama `html_docs` y que no hay ningún otro directorio que empiece de modo igual o parecido. Escribiremos:

```
$ cd html (presionamos el tabulador aquí)
# (entonces la línea se nos queda:)
$ cd html_docs/
```

Si pulsamos INTRO, nos habremos cambiado al directorio `html_docs/`. Mira lo que acabamos de descubrir. Si estamos escribiendo un nombre de archivo o de directorio y presionamos el tabulador, si ese nombre existe, BASH lo rellenará por nosotros. Cómodo, ¿no? De esta forma no tendremos que escribirlo entero. Ahora supongamos que además de `html_docs` hay otro directorio al mismo nivel que se llama `html_includes`. Si escribimos ahora **`cd html`** y presionamos el tabulador, no habrá salida porque no sabrá cuál de ellos elegir. La solución es pulsar el tabulador dos veces y nos mostrará las opciones que coinciden con lo que llevamos escrito, con lo cual podresmo seguir escribiendo hasta que hayamos eliminado una de las opcioens y pulsemos el tabulador de nuevo solo una vez, entonces sí que nos rellenará el nombre de archivo o directorio. Ten en cuenta que esto sirve con casi cualquier comando y es indiferente para referencias relativas al directorio actual o absolutas del sistema de archivos , es muy útil y permite ahorrar mucho tiempo y equivocaciones escribiendo nombres de ficheros o directorios en la línea de comandos. Sin duda, una importantísima característica para los que no nos sobra el tiempo :-)

Ya hemos dado un repasito a algunas cosas básicas de la terminal BASH. Por lo menos ya no estás perdido, practica todo esto un poco, y también léelo de vez en cuando, porque conforme vayas practicando, mejor comprenderás algunas cosas que ahora no eres capaz de comprender o asimilar sin haber visto otras primero. El dominio de la terminal BASH en un sistema Linux es imprescindible, porque (afortunadamente) no hay GUIs (interfaces de usuario gráficas) para todo lo que podemos hacer en nuestro sistema. Verás como aunque estás trabajando en modo gráfico dentro de unas horas o unos días, tendrás abierta casi siempre una ventan de la terminal en modo gráfico para hacer todas esas cosas que sólo es posible (y más eficiente) hacerlas desde una terminal o que son más flexibles desde la línea de comandos.

Aunque más tarde seguiremos viendo cosas más avanzadas de BASH, como la personalización del prompt (le podremos dar colores) o la asignación de variables de entorno que podamos necesitar... y mucho más.

Capítulo 4. VIM básico

Conceptos básicos

A lo largo de la administración de un sistema Linux podemos toparnos con la necesidad de editar un fichero de configuración o algo parecido. Bien, en Linux hay muchos editores que pueden satisfacer nuestras necesidades (nano, pico, jedit...). En este capítulo *NO* vamos a cubrir una guerra típica "VIM vs. Emacs". Simplemente mostraremos un editor muy potente y queremos que con estas nociones cualquier usuario pueda beneficiarse de la potencia de VIM sin tener que entrar en muchos detalles ni en cosas complicadas.

Antes de nada vamos a introducir un poco sobre qué es VIM, de dónde viene y qué vamos a aprender en este capítulo.

VIM es un editor de textos para Linux. Es un derivado de VI, VIM significa VI iMproved, es decir, el mismo editor VI pero mejorado. Y en este capítulo aprenderemos a hacer un uso básico de este fabuloso editor.

¿Cómo funciona VIM?

VIM tiene dos grandes modos para funcionar, el modo de edición y el modo comandos. Para pasar del modo comandos (que es en el que se inicia VIM) al modo edición, pulsamos la tecla **I** o **Insert**. Para pasar del modo edición al modo comandos pulsaremos la tecla **Esc**.

Para iniciar VIM, haremos lo siguiente:

```
$ vim <fichero>
```

VIM iniciará en el ’Modo Edición’ que explicaré en la siguiente sección.

Modo Edición

Este es el modo en el que podremos editar un texto, o crear uno nuevo... Si pulsamos la tecla **I** o **Insert** otra vez cambiaremos el modo Insert por el modo Replace. No creo que haya que explicar mucho las diferencias entre insertar texto o reemplazar el texto. Para movernos por el fichero podemos utilzar las flechas,**Re.Pág** y **Av.Pág**.

> **Nota:** Para volver al modo comandos, sirve con pulsar la tecla **Esc**.

Modo comandos

Este es el modo más interesante de VIM, con él podemos hacer cosas muy curiosas, como por ejemplo, activar el resaltado de sintaxis, deshacer, abrir un nuevo fichero, guardar, etc..

En la siguiente tabla explicaré los comandos que pueden resultar más interesantes.

Tabla 4-1. Comandos más usuales en VIM

Comando	Descripción	Ejemplo de uso (si aplica)
:w	Guarda el buffer en el fichero	--
:w [fichero]	Guarda el buffer en fichero (como un Save as...)	:w ~/ficherito
:q	Salir de VIM	--
:q!	Salir de VIM sin guardar los cambios	--
:wq	Salvar y guardar	--
:u	Deshacer	--
Ctrl-R (^R)	Rehacer	--
:d ó :dN	Borrar la línea corriente o borrar N líneas a partir de la actual.	--
:syntax [on | off]	Activa/Desactiva el resaltado de sintaxis	:syntax on
:s/[patrón]/[reemplazo]/g	Sustituye [patrón] por [reemplazo] en la línea actual.	s/hoal/hola/g
:![comando] [argumentos]	Ejecuta [comando] pasándole los argumentos [argumentos] en el shell	:!gcc -g -O0 -o prueba prueba.c
:+[número]	Baja [número] líneas	:+256
:-[número]	Sube [número] líneas	:-12
:[número]	Va a la línea [número]	:1207
:+/[patrón]	Busca [patrón] en el documento	:+/donde te metes

El modo especial: VISUAL

Para entrar a este modo tenemos que pulsar **v** en el modo comandos. Para salir de el pulsaremos **Esc**.

El modo VISUAL nos permitirá eliminar grandes bloques de texto sin tener que ir borrando línea a línea. Para ello entraremos en el modo VISUAL (pulsando **v**) y nos moveremos con los cursores para seleccionar el bloque que queremos eliminar. Una vez tenemos seleccionado el bloque, pulsamos **Supr** y ya está, VIM nos informará de que hemos eliminado un bloque de "n" líneas.

~/.vimrc

Vim cuenta con un fichero de configuración en ~/.vimrc que nos puede ser muy util para personalizar nuestro VIM. Cualquier comando de VIM se puede introducir en él. Además de esto, hay muchas opciones que pueden cambiarse y hasta sentencias condicionales (algo que queda fuera del alcance de este capítulo). Cualquier línea que comience con " (comillas) será ignorada por VIM. A esto lo llamamos comentarios.

Como ejemplo mostraremos como habilitar el resaltado de sintaxis para todos los ficheros que abra el lector:

```
" " "
" ~/.vimrc
" " "
"
" Activamos el resaltado de sintaxis
syntax on

" Activamos la barra de informacion
set ruler

" Desactivamos la compatibilidad con VI
set nocompatible

" El identado lo hace VIM
set autoindent

" Características especiales de VIM
if has("autocmd")
 " Habilitamos características propias de cada tipo de fichero
 filetype plugin indent on
 " Volvemos a la última línea que habíamos editado.
 " ( Juntar las 2 siguientes líneas en una )
 autocmd BufReadPost * if line("'\"") > 0 &&
line ("'\"") <= line("$") | exe "normal g'\"" | endif
endif " has("autocmd")
```

Obtener ayuda

Obviamente esta ha sido una introducción muy liviana a Vim. Pero ¡VIM es mucho más!. Y el mejor punto de partida es ejecutar **vimtutor**. Este comando lo que hace es ir guiándonos paso a paso y con ejemplos. Editando buffers (ficheros) preparados para el correcto aprendizaje de vim. Vim es muy grande, se pueden hacer muchas cosas con él. Es muy recomendable utilizar **vimtutor** para iniciarse en este gran editor.

Capítulo 5. Usuarios y Grupos. Permisos

En este capítulo abordaremos un tema escabroso en Linux (quizas le interese a los paranóicos). Permisos, usuarios y grupos como bien indica el título. Un *sistema mal administrado* podría ser, y de hecho es un auténtico coladero. Ante todo quisiera por favor que reportéis los posibles fallos tanto de esta sección como de cualquier otra, ya que este manual pretende ser un documento útil, por lo que os doy las gracias por adelantado. Vamos a ello.

Usuarios

Linux es un sistema multiusuario por lo que es necesario la administración (segura) de los distintos usuarios que van a hacer uso de los recursos del sistema. De momento no vamos a dar grandes nociones de seguridad, pero sí una introducción a los tipos de usuarios y qué privilegios deben tener.

Dentro de un sistema existen al menos 3 tipos de usuarios.

- *Usuarios normales* con más o menos privilegios que harán uso de los recursos del sistema. Son generalmente inexpertos y propensos a causar problemas. Simplemente deben poder usar algunos programas y disponer de un directorio de trabajo.
- *Usuarios de Sistema* son aquellos encargados de los demonios del sistema, recordemos que para *Linux* todo es un fichero, el cual tiene un dueño y ese dueño tiene privilegios sobre él. Así, es necesario que algún usuario del sistema *posea* los procesos de los demonios, como veremos más adelante.

  Resumiendo, es necesario que para algunos servicios del sistema se creen usuarios (generalmente para demonios). Como puede ser el caso de Mail, irc... Estos usuarios tendrán los privilegios necesarios para poder hacer su tarea, gestionar estos usuarios es de gran importancia. No obstante, este tipo de usuarios no necesita que se le asigne un shell, puesto que simplemente poseerán los demonios, pero no será necesario que hagan `login` en el sistema.

- *ROOT* Este es *dios* ;-) Como ya habéis leido en capítulos anteriores cuidado con lo que se hace al entrar en el sistema como root. Él lo puede todo, en principio no hay restricciones para ÉL (aunque algunos programas nos avisarán de que estamos haciendo una auténtica burrada desde el punto de vista de la seguridad).

Además del criterio anterior, existe un criterio secundario para clasificar a los usuarios de un sistema. Ya vamos conociendo la tendencia de los UNIX y Linux, primero, por hacer que casi todo sea un fichero, y segundo, por hacer que absolutamente todo tenga un dueño (como iremos descubriendo a medida que usamos el sistema, nada se deja al azar en este aspecto), incluidos los procesos que se están ejecutando en el sistema. Otro concepto que lleva a esta segunda clasificación es, el de los usuarios de los servicios del host. Este tipo de usuarios simplemente accede remotamente a algunos servicios de nuestra máquina, tales como correo-electrónico o FTP. Esto nos lleva a definir:

- *Usuarios de login*. Estos primeros, pueden hacer login en el sistema y usar una shell en él. Es decir, tienen una shell válida de inicio (como se indica más abajo), generalmente, `/bin/bash`.
- *Usuarios sin login*. Este tipo de usuarios, bien son usuarios de sistema, o bien usuarios de los servicios del host. En ambos casos no pueden hacer login en el sistema directamente ni usar un shell. En otras palabras, no tienen asignado un shell válido. Su misión, es "poseer" algunos archivos y directorios del sistema, y manejarlos restringidamente a través de algunos programas (el servidor

FTP, el servidor de correo electrónico, etc. darán a los usuarios de los servicios del host los privilegios suficientes: almacenar mensajes, etc.). En el caso de los usuarios de sistema, poseerán además algunos procesos (en el capítulo de procesos aprenderemos más acerca de este sub-tipo especial de usuarios).

Administración de Usuarios.

La administración de usuarios se realiza en todas las distribuciones de manera muy parecida, dada la herencia *UNIX*, aunque en el fondo todas hacen lo mismo. Según la política que lleven lo pueden realizar de una manera u otra, por lo que aquí veremos la forma la forma interna de trabajar de los programas a la hora de añadir o quitar usuarios, y al final expondremos ejemplos concretos de las distros más conocidas.

¿Cómo añadir un usuario al sistema? Hay que seguir una serie de pasos que relatamos a continuación. Pero antes veremos la estructura de los archivos que vamos a tocar.

- `/etc/password` - Archivo que mantiene la base de datos de los usuarios del sistema y tiene la siguiente forma:

  ```
  nombre_de_usuario:password(si es shadow será
  x):uid:gid:comentario:home_del_usuario:shell
  ```

  Estos campos son:

  - `Nombre de Usuario` - Es el nombre con el que entrará en el sistema.
  - `Password` - La palabra de paso necesaria para entrar (cifrada). Si nuestro sistema usa shadow (explicado después), este campo será una *x*
  - `UID` - (User ID) Número que lo identifica en el sistema, recordemos que los ordenadores se llevan mejor con los números.
  - `GID` - (Group ID) Número que identifica al grupo principal al que pertenece el usuario.
  - `Comentario` - Opcional, si es necesario aclarar algo, esto solo es para el administrador, pues el sistema no lo usa.
  - `home_del_usuario` - Ruta absoluta del directorio de trabajo del usuario.
  - `Shell` - Intérprete de comandos del usuario, que será el que use inmediatamente después de entrar en el sistema, por defecto es `/bin/bash`. Para usuarios sin login, aunque puede que ahora no lo necesites, la shell no válida típica para poner en este campo es `/bin/false`.

- `/etc/group` - Archivo de los grupos del sistema; de su administración y uso hablaremos en el siguiente apartado. El archivo tiene la siguiente estructura:

  ```
  nombre_grupo:password:GID:lista_usuarios
  ```

  - `Nombre del Grupo` - Por defecto con los comandos habituales se crea un grupo con el mismo nombre que el usuario creado, aunque pueden existir otros grupos con nombres específicos.
  - `password` - Se usa para dar a una serie de individuos un mismo directorio con una cuenta común.
  - `GID` - (Group ID) Número de Identificación en el Sistema del grupo.
  - `lista de usuarios` que pertenecen al grupo, separados por comas.

- `/etc/shadow` - Para sistemas que usen shadow, que no es más que una medida de seguridad. Los sistemas que no usan shadow guardan el password en `/etc/passwd` pero este archivo tiene la peculiaridad de que debe ser legible por todo el mundo, si no, no podría ni hacerse un *ls*. Este archivo podría caer en manos de un usario ilegítimo y este ejercer técnicas de crackeo sobre las claves. Como solución del problema lo que se hace es almacenar todos los datos de los usuarios en el `/etc/password` menos sus contraseñas; que se almacenan en `/etc/shadow`, el cual sí tiene restringidos los permisos y no es accesible por los usuarios normales.

  ```
  usuario:password:días del último cambio: dias antes del cambio:Dias
  despues del cambio: tiempo de aviso:dias antes de la inhabilitacion:
  perido que lleva caducado:reservado:
  ```

  - `Usuario` - Nombre del usuario
  - `password` - Aquí sí, es el password cifrado.
  - `Tiempo del último cambio de password` - Pero el tiempo cuenta desde el 1 de enero de 1970, comienzo de la era UNIX.
  - `Días antes del cambio` - Periodo (en días) donde el password debe ser cambiado.
  - `Dias despues del cambio` - En los días después donde debe ser cambiado.
  - `Tiempo del aviso` - Periodo en el que el sistema tiene que avisar de la necesidad del cambio.
  - `Inhabilitación` - Días antes de la inhabilitacion de la cuenta.
  - `Perido caducado` - Días desde el 1 de enero de 1970 en el que la cuenta está deshabilitada.
  - `Campo reservado`

Bien ahora, ya que conocemos la estructura de los archivos, creemos un nuevo usuario, sólo tendremos que usar un editor, que por razones de seguridad son **`vipw`** para el archivo `/etc/passwd` y **`vigr`** para editar `/etc/group`; ambos usan como editor el que esté en la variable `$EDITOR` del sistema, y `passwd` para crear el password. Sigamos los siguientes pasos:

1. Lo primero es entrar como root

   ```
   $ su -
   password:
   # vipw /etc/passwd
   ```

2. Antes de nada, el comando `su` se explica al final de este capítulo. Ya estamos editando el fichero `/etc/passwd`, ahí estarán las lineas de otros usuarios que ya estén creados. Esto que sigue es un ejemplo.

   `prueba:x:1005:1005::/home/prueba:/bin/bash`

   Hemos escrito un nombre cualquiera (prueba), el password le ponemos *x* dado que es un sistema con shadow. Si queremos crear una cuenta sin contraseña, en este campo en vez de la x no pondremos nada. ADVERTENCIA: Esto es un considerable riesgo de seguridad. Incluso una cuenta de usuario puede usarse para hallar información útil para posteriormente poder atacar un sistema.

3. Ahora hay que tocar en `/etc/group` para crear el grupo del usuario (todo usuario tiene un grupo principal), le damos el mismo nombre que al usuario, y el GID que hemos puesto antes, el 1005. Por supuesto, tanto los UID como los GID no pueden estar repetidos, así que nos aseguramos de coger uno que no esté ya cogido. Entonces, hacemos:

   ```
   # vigr /etc/group

   prueba:x:1005:
   ```

4. Editamos `/etc/shadow` :

   ```
   prueba:!:12173:0:99999:7:::
   ```

   Bueno lo peor será calcular la fecha de creación pero sino randomized() :-). Notad que en el campo password le hemos puesto *!* porque ahora le daremos un password con **`passwd`**.

5.

   ```
   # passwd prueba ❶
   Enter new UNIX password: ❷
   Retype new UNIX password: ❸
   ```

   ❶ Ejecutamos passwd y pasamos el nombre del usuario como argumento.

   ❷ Introducimos el password para ese usuario, hay que darse cuenta de que a medida que escribimos, por razones obvias de seguridad no se irá escribiendo nada.

   ❸ Repetimos la password

   Si no hay error, listo.

6. Ahora le tenemos que crear el directorio de trabajo, por convenio los directorios de trabajo de los usuarios normales están bajo `/home`, y suelen ser `/home/usuario`.

   ```
   # mkdir /home/prueba
   ```

7. Copiamos los archivos de inicio desde `/etc/skel` , que contiene principalmente archivos de configuración por defecto.

   ```
   # cp /etc/skel/.* /home/prueba
   ```

8. Y por último, hay que hacer que tome posesión de lo que es suyo:

   ```
   # chown prueba.prueba -R  /home/prueba
   ```

   Este último comando se explica más adelante. Ahora sólo falta hacer:

```
# su - prueba
$
```

También, en vez de usar `su`, podemos hacer login en el shell directamente con el nuevo usuario creado para comprobar que lo hemos hecho correctamente.

Esto es lo que hace cualquier Linux para crear una cuenta. La forma de borrar un usuario es igual de fácil, borrándolo en `/etc/passwd`, `/etc/group` y en `/etc/shadow`. Cualquier rastro del usuario debería también ser eliminado, además de que sería bastante recomendable (si no necesario) hacer una busqueda de los archivos que el usuario mantiene. Esta búsqueda podría ser:

```
# find / -uid uid_del_usuario > archivos_del_usuario
```

De esta manera los podemos guardar, revisar y hacer lo conveniente con los archivos. Más sobre `find` en secciones posteriores.

A continuación veremos los comandos que hacen esto mismo y alguna diferencia entre las distintas distribuciones.

Comandos de Administración

Estos comandos nos permitirán crear, borrar y modificar las cuentas de usuarios en el sistema.

adduser y useradd son dos comandos que hacen prácticamente lo mismo. Nota que el estándar es `useradd`, adduser puede, en algunas distribuciones ser un simple enlace a useradd, o simplemente no existir. El comportamiento por defecto de `useradd` es muy subjetivo dependiendo de la distribución de Linux que estemos usando. Así, mientras que en algunas distribuciones hará casi todos los pasos anteriores por nosotros, en otras sólo añadirá el usuario a `/etc/password` y `/etc/shadow`, teniendo que realizar nosotros los pasos restantes. Aun así, las opciones que recibe useradd responden igual en todas las distros. Como ya hemos hecho con algún otro comando, puedes ver las opciones de `useradd` mediante **`man useradd`**, moverte con las teclas de dirección y volver al shell pulsando la tecla Q.

Las opciones más comunes para **`useradd`** son:

```
# useradd -g users -d /home/usuario -s /bin/bash -m -k /etc/skel usuario
```

- `-g` Indica cuál es el grupo principal al que pertenece el usuario; en este caso, `users`
- `-d` Establece el que será el directorio de trabajo del usuario, por convenio para usuarios normales, es `/home/nombre_de_usuario`
- `-s` Es la shell por defecto que podrá usar el usuario después de hacer login. Lo normal es que sea `/bin/bash`, esto es, el shell bash.
- `-m -k` Estas dos opciones se complementan. La primera, hace que se cree el directorio de trabajo del usuario en el caso de que este no exista. La segunda, copia los ficheros del directorio especificado al que se ha creado del usuario.
- *usuario* Por último, *usuario* es el nombre del usuario que estamos creando.
- *-G* Aunque esta opción no está en el comando de ejemplo que hemos puesto arriba, es muy útil, puesto que permite especificar la lista de grupos a la que también pertenecerá el usuario aparte del

grupo principal. Por ejemplo `-G audio,cdrom,dip` añadido en el comando anterior haría que *usuario* perteneciese a estos grupos además de al suyo principal.

Siempre podemos matizar detalles en la creación de usuarios editando los ficheros de configuración que ya conocemos.

La cuenta del usuario no estará activada hasta que no le asignemos un password. Esto lo podemos hacer vía **`password nombre_de_usuario`** (para cambiar el password procederíamos exactamente de la misma forma). Cualquier usuario normal puede cambiar su password cuando haya entrado al sistema, bastando en este caso con escribir el comando **`password`** sin argumentos ni opciones.

Para desactivar una cuenta de usuario sin necesidad de borrala del sistema, podemos editar `/etc/passwd` y cambiar la `x` por un `!` en el campo del password. La cuenta se habilitaría asignando un password con **`password`** o bien volviendo a poner la `x` que había, quedando la cuenta entonces con el mismo password que tenía antes de deshabilitarla.

Podríamos, si nuestra distro dispone de ellas, utilizar las herramientas interactivas para crear usuarios (a veces incluso existen interfaces gráficas de ventanas para gestionar usuarios y grupos, solo accesibles por root); generalmente `adduser`. Aunque pueden resultar más cómodas, siempre es conveniente saber el método general de creación de usuarios, de tal forma que esos conocimientos nos servirán para cualquier distribución de Linux que vayamos a usar.

En algún momento podríamos necesitar editar una cuenta de usuario creada, o borrarla. Existen también comandos de administración estándares para esto; son **`usermod`** y **`userdel`**. Puedes ver sus páginas del manual con `man`. Aunque, una vez más, editando los ficheros de configuración no necesitarás memorizar opciones para efectuar esos cambios.

El comando **`whoami`** muestra el nombre de usuario que está dentro del sistema usando el shell desde el que se le llama.

Grupos

En la administración de grupos no vamos a gastar muchas energías ya que no es, en un nivel básico, algo excesivamente complejo.

Los *grupos* es una manera en los sistemas multiusuario como *Linux* de otorgar una serie de privilegios a un conjunto de usuarios sin tener que dárselo de forma indivial a cada uno.

El fichero encargado de mantener los grupos del sistema es `/etc/group` y también hemos visto su estructura. Por lo que veremos los comandos que añaden, quitan y modifican los grupos; así como notas generales en la gestión de grupos.

Hemos dicho que todo usuario tiene siempre un grupo principal al que pertenece. Hay dos posibilidades para los usuarios normales: que todos tengan el mismo grupo principal (generalmente `users`) o que cada usuario tenga un grupo principal específico (casi siempre del mismo nombre que el usuario). Esto responde a las necesidades de cada sistema. En el primer caso, los directorios de trabajo de los usuarios suelen ser accesibles por el resto de usuarios (no es lo más común); mientras que en el caso de que cada usuario tenga un grupo principal, lo normal es que los directorios de trabajo de cada usuario sean sólo accesibles por ese usuario (que sí es lo más común).

La utilidad del grupo principal de un usuario se entenderá mejor cuando lleguemos a los permisos.

Además, en el sistema hay más grupos que los principales de cada usuario. La misión de estos otros grupos es la de otorgar unos permisos similares al conjunto de usuarios que forman parte de él ante un directorio, un archivo, un dispositivo, etc.

Es muy común la necesidad de dar a unos cuantos usuarios permisos para que puedan, por ejemplo, leer los documentos de un directorio determinado (por ejemplo, informes de una empresa), al tiempo que al resto de usuarios no. Así, podríamos crear un grupo llamado *contables*, y agregar los usuarios que son contables a este grupo. Después, haríamos pertenecer (como veremos en Permisos) el directorio mencionado a este grupo, y le daríamos permisos de lectura para el grupo.

Además de para compartir archivos o directorios entre varios usuarios, existen grupos como `audio`, `cdrom`, y similares. Dispositivos como la tarjeta de sonido, el grabador de CDs, etc. tienen como usuario "dueño" a root, y como grupo "dueño" a uno de estos grupos. Así, para cada usuario que queramos que pueda usar la tarjeta de sonido, debemos añadirlo al grupo `audio`.

Administración de grupos

Ya hemos aprendido cómo establecer el grupo principal de un usuario, y cómo hacer a la hora de su creación que pertenezca a otros grupos adicionales; así como el fichero `/etc/group`. Así que ya no queda mucho en este aspecto.

Para añadir un usuario a un grupo de forma manual, todo lo que hay que hacer es editar `/etc/group` y añadir al usuario a la lista del último campo. Si ya hay usuarios, éstos se separan con comas.

Si lo que queremos es crear un nuevo grupo de forma manual, la cosa es igual de sencilla. Al igual que ya hicimos agregando el usuario *prueba* de forma manual, añadiremos otra entrada a este fichero con el nombre del grupo, la x en el password, un GID que no esté siendo usado y la lista de usuarios detrás.

También tenemos comandos que hacen esto mismo: **`groupadd`** , **`groupdel`** y **`groupmod`**.

`groupadd` sirve para crear un nuevo grupo:

```
# groupadd -g gid grupo
```

La opción `-g` va seguida del Group ID (numérico) que asignaremos al grupo, y *grupo* es el nombre del grupo creado. Si no indicamos la opción `-g`, el sistema seleccionará por nosotros un número GID que no esté siendo usado.

Lo mismo que sabemos ya sobre la edicición y borrado de usuarios es aplicable a los grupos. Lo más fácil es editar el fichero `/etc/group` directamente; recordando que si quitamos un grupo, ningún usuario podrá tener ese grupo como su grupo principal; y además que tendremos que cambiar los permisos (después más en esto) de los archivos que perteneciesen a este grupo pues de no hacerlo los estaríamos dejando asignados a un GID inexistente para el sistema, y esto no es recomendable.

Un comando curioso es **`groups`**, que mostrará una lista de grupos a los que el usuario actual pertenece.

Permisos y dueños

Todos y cada uno de los elementos del sistema `/` de Linux tienen dueño, ya sean ficheros, directorios, o enlaces a dispositivos. Por un lado, tienen un *usuario* dueño, y por otro, un *grupo* dueño. El usuario y el grupo que son dueños de un elemento no tienen por qué guardar una relación del tipo que el usuario debería estar dentro del grupo o cosas por el estilo. Son totalmente independientes. Así, puede existir un fichero que tenga como usuario propietario a `username`, un usuario normal, y tener como grupo propietario al grupo `root`.

Cuando se trabaja en el sistema, los programas "hacen dueños" de los ficheros creados durante la sesión al usuario de esta sesión y a su grupo principal por defecto; aunque esto puede cambiarse. Es lógico, que los ficheros que estén bajo el directorio de trabajo de un usuario le pertenezcan.

Siempre que tratemos con permisos y con dueños de elementos, debemos tener siempre presente el hecho de que el sistema de ficheros de Linux es *jerárquico*; esto implica que los cambios que hagamos, por ejemplo, en un directorio, pueden influir en el resto de elementos que están contenidos en un nivel inferior a éste (los archivos que contiene, los directorios que contiene, lo que contienen esos otros directorios, y así sucesivamente).

Con un simple **`ls -l`** en cualquier parte del sistema, podemos ver en la forma `usuario grupo` los dueños de cada elemento que sale en el listado largo. Entonces ya sabemos cómo comprobar esto.

El comando **`chown`** (CHange OWNer - cambiar propietario) permite cambiar el propietario de los elementos del sistema de archivos. Pero es lógico que si somos un usuario normal no podremos cambiar de propietario los elementos que pertenecen a root o a otros usuarios. En cambio, como root podremos cambiar el propietario de cualquier cosa. Aquí describimos las opciones más usadas de este comando, pero puedes ir mirando su página del manual del sistema.

```
# chown usuario elemento(s)
# chown usuario.grupo elemento(s)
```

En el primero de los dos comandos anteriores, el usuario dueño de *elementos(s)* será cambiado a *usuario*. El grupo deuño de *elemento(s)* se conservará el que estuviera antes de introducir este comando.

Con respecto al segundo comando, actúa exactamente igual que el anterior, con la pequeña diferencia que también cambiará el grupo dueño de *elemento(s)* pasando a ser *grupo*. Si sólo queremos cambiar el grupo de un elemento o lista de ellos, podemos usar el comando **`chgrp`**.

```
# chgrp grupo elemento(s)
```

elemento(s) puede ser una lista de archivos y directorios, o simplemente uno de ellos. Podemos usar los wildcards conocidos (como por ejemplo el asterisco: `*` para indicar varios archivos con una sola expresión. La importante opción **`-R`** permite cambiar dueños de directorios y de todo lo que tengan debajo, es decir, recursivamente:

```
# chown -R usuario.grupo directorio/
```

Este comando cambiará el usuario y grupo dueños tanto de *directorio/* como de todo lo que contenga hasta cualquier nivel, es decir, todo lo que esté "debajo" de *directorio*, y el directorio mismo cambiarán de dueño.

¿Y para qué todo esto de los dueños de archivos y directorios? Para poder asignar permisos adecuadamente.

Un archivo tiene distintos niveles de permisos: lectura, escritura y ejecución. Los permisos sobre un archivo (o directorio) pueden ser distintos para el usuario dueño, para los usuarios pertenecientes al grupo dueño, y por último para el resto de los usuarios del sistema. Así, podemos hacer que el usuario dueño puede leer, escribir, y ejecutar un fichero; que el grupo dueño solo pueda leerlo, y que el resto de usuarios del sistema no tengan ningún permiso sobre él, por ejemplo.

Una buena asignación de dueños de elementos junto con una política adecuada de permisos sobre estos elementos, permiten obtener dos cosas: un sistema multiusuario, y un sistema seguro.

Si haces un `ls -l` en un directorio que tenga algunas cosas verás algo como:

```
$ ls -la bin/

drwxr-xr-x   2 root     root         4096 Apr 16 17:19 .
drwxr-xr-x  21 root     root         4096 May 23 20:34 ..
-rwxr-xr-x   1 root     root         2872 Jun 24  2002 arch
-rwxr-xr-x   1 root     root        94364 Jun 25  2001 ash
-rwxr-xr-x   1 root     root       472492 Jun 25  2001 ash.static
-rwxr-xr-x   1 root     root        10524 Jul 19  2001 aumix-minimal
lrwxrwxrwx   1 root     root            4 Feb  4  2002 awk -> gawk
```

Fíjate en el campo de más a la izquierda del listado. Podemos ver como cuatro grupos. El primero es de un caracter solamente. Este caracter es una `d` si el elemento listado es un directorio, una `l` si el elemento es un enlace, y un guión `-` si el elemento es un archivo normal.

A continuación hay tres grupos. Cada uno de estos grupos tiene tres letras, pudiendo ser estas `rwx` o pudiendo ser sustituidas en algún caso por un guión. El primero de estos grupos indica los permisos que tiene sobre el elemento listado su usuario dueño; el segundo grupo indica los permisos que tienen sobre el elemento los usuarios que pertenezcan al grupo dueño, y el tercer grupo indica los permisos que tienen sobre el elemento el resto de usuarios del sistema.

En el caso de un archivo o un enlace (sobre los que hablaremos posteriormente), la `r` en cualquiera de estos "grupos" indica que se tienen permisos de lectura sobre el elemento. La `w` indica que se tienen permisos de escritura sobre el elemento, y la `x` indica que se tienen permisos de ejecución sobre el elemento. Un guión sustituyendo a cualquiera de estas letras indica que no se tiene el permiso al que está sustituyendo. Así, veamos algún ejemplo del listado anterior:

```
-rwxr-xr-x   1 root     root         2872 Jun 24  2002 arch
```

Es un archivo porque su primer caracter es un guión. Su usuario dueño es root, y su grupo dueño es el grupo root también. root tiene todos los permisos sobre él: `rwx`, esto quiere decir que puede leer el archivo arch, escribir en él y ejecutarlo. El grupo root sólo lo puede leer y ejecutar, y el resto de usuarios del sistema, también sólo pueden leerlo y ejecutarlo.

El caso de los directorios es un poco distinto. Los permisos `rwx` para un directorio, indican: la `r` y la `x` para el caso de un directorio difícilmente se entienden separadas. Son necesarias para que un usuario pueda "examinar" ese directorio, ver lo que tiene y navegar por él. La `w` indica que el usuario que posea este permiso puede colocar nuevos archivos en este directorio; así como también borrarlo.

Lo más común es que los directorios que deban poder ser "examinados" por todos los usuarios tengan permisos `r-x` en el tercer grupo de permisos. Pero con estos permisos no podrán colocar nada dentro de ese directorio, aunque sí podrán hacerlo dentro de un directorio de nivel inferior en el que sí tengan permisos de escritura. Ten en cuenta que si tenemos por ejemplo un directorio llamado `superior/` y dentro de éste tenemos un directorio llamado `personal/`, y que un usuario tienen permisos de escritura en este segundo directorio, que es de nivel inferior, para poder acceder a él y escribir, debe este usuario poseer, como mínimo, permisos `r-x` en el de nivel superior, esto es, en `superior/`. Por otra parte, esto es absolutamente lógico: ¿cómo va a poder escribir un usuario en un directorio si no puede llegar hasta él? Esto mismo también se aplica para la lectura. Por ejemplo, el servidor web no podrá servir un directorio a la Internet si no dispone de permisos `r-x` para los directorios superiores a él.

Para cambiar los permisos de los elementos del sistema de ficheros, usamos el comando **`chmod`**.

```
# chmod -R ABC elemento
```

La opción -R es opcional, y cumple exactamente la misma función que en el comando `chown`. A B y C son un número de una cifra respectivamente. El primer número representa los permisos que estamos asignando al usuario dueño , el segundo los del grupo dueño, y el tercero los del resto de usuarios. Cada una de las cifras posibles corresponde con los permisos del usuario en binario; aunque es más fácil aprenderse qué hace cada cifra que pasar la cifra a binario cada vez que queramos cambiar los permisos a algo. Algunos ejemplos:

- El `4` en binario es `100`, por tanto, los permisos que otorga son `r--`, esto es, sólo lectura.
- El `5` en binario es `101`, por tanto, los permisos que otorga son `r-x`, lectura y ejecución.
- El `6` en binario es `110`, por tanto, los permisos que otorga son `rw-`, lectura y escritura.
- El `7` en binario es `111`, por tanto, los permisos que otorga son `rwx`, lectura, escritura y ejecución.

Los permisos de ejecución sólo se otorgarán a programas o scripts (con los que trataremos después); ya que hacerlo a los archivos normales carece por completo de sentido. Así, un comando de ejemplo podría ser:

```
$ chmod 640 mitexto
```

Este comando asignaría permisos de lectura y de escritura al usuario propietario, y permisos de lectura a los usuarios del grupo dueño, y ningún permiso al resto de usuarios para el archivo `mitexto`. Puedes ir haciendo pruebas combinando los distintos números y ver los permisos que otorgan mediante **`ls -l`**. Recuerda que los directorios que quieras que puedan ser "examinados", debe tener permisos de "ejecución" por parte de los usuarios que quieras que puedan acceder a ellos, por ejemplo podrías asignarlos con el número 5. A los que además quieras que puedan crear archivos en ese directorio, podrías darle esos permisos mediante el número 7. Con la opción `-R` puedes hacer que los permisos se asignen de modo recursivo a un directorio y a todo lo que hay debajo de él.

Un modo muy común para los directorios que deban ser accesibles por todo el mundo es 755, de forma que el usuario dueño pueda además escribir. Los directorios `/home/usuario` suelen tener permisos 750 para que el resto de usuarios no puedan acceder al directorio de trabajo de un usuario.

> **Aviso**
>
> Una mala asignación de permisos puede dar lugar a ataques locales, si dejamos a algunos usuarios permisos para modificar partes importantes del sistema de ficheros.
>
> Ten cuidado cuando cambies permisos, sobre todo si eres `root`.

Un modo muy común de añadir permisos de ejecución a un archivo (generalmente un *script*) para todos los usuarios del sistema, sin tener que estar recordando qué números otorgan permisos de ejecución, es usar la opción `+x` de `chmod`, por ejemplo:

```
$ chmod +x mi_script.sh
```

Esta forma de asignar permisos es extensible, y según los casos, más sencilla que la de los números. En general es así:

```
$ chmod ABC fichero
```

Donde A es `u` (usuario), `g` (grupo) o bien `a` (todos). Cuando es `a`, se puede omitir.

B es `+` o bien `-` , indicando el primero añadir un cierto permiso y el segundo quitarlo.

C es `r` (lectura), `w` (escritura) o bien `x` (ejecución).

Ejemplos:

```
$ chmod g+w fichero
$ chmod -r fichero
$ chmod u+x fichero
```

El primer comando otorga permisos de escritura sobre `fichero` a los usuarios del grupo al que el fichero pertenece.

El segundo comando elimina los permisos de lectura sobre `fichero` a todo el mundo.

El tercer comando da al usuario dueño de `fichero` permisos de ejecución.

El comando su

El comando **`su`** (Set User) está relacionado con el login en el sistema y con los permisos. El uso principal de este comando es que un usuario normal adquiera los permisos de otro usuario del sistema (incluido root) siempre y cuando sepa su password.

Es muy común que, si somos nosotros el "dueño" de la contraseña de root, y por tanto la persona encargada de la administración del sistema, trabajemos normalmente como usuario normal por motivos de seguridad. Pero podemos necesitar convertirnos en root para alguna tarea específica: reiniciar el servidor web, modificar la configuración del sistema... para después volver a "ser" nuestro usuario normal.

```
$ su
Password: [ Escribimos el password de root ]
#
```

`su` llamado "a secas" como en el ejemplo anterior asume que el usuario actual quiere adquirir los permisos de root. Si proporcionamos el password adecuado ya los tendremos. Podemos ahora hacer las tareas de administración que necesitemos. Escribiendo **`exit`** volveremos a "ser" nuestro usuario normal.

Hay gente que considera útil tener siempre "en segundo plano" una shell con permisos de root. Esto se puede conseguir con los comandos **`suspend`**, **`fg`** y **`jobs`**. Por ejemplo:

```
$ su
Password: [ Password de root ]
# suspend ❶
[1]+  Stopped                 su ❷
$
$ fg %1 ❸
#
```

❶ Detenemos la shell con permisos de root y la dejamos "esperando".

❷ Nos informa de que se detuvo, y le asigna el número de referencia 1 para recuperarla posteriormente.

❸ Recuperamos la shell con permisos de root.

Mediante `suspend` podemos detener la shell con permisos de root que arrancamos antes mediante `su` y recuperarla cada vez que necesitemos permisos de root sin necesidad de teclear el password de nuevo. Escribiendo **`exit`** en la shell con permisos de root podríamos abandonarla definitivamente.

`su` nos permite también adquirir los permisos de otros usuarios del sistema siempre que tengamos su password:

```
usuario@maquina $ su otrousuario
Password: [ Password de "otrousuario" ]
otrousuario@maquina $
otrousuario@maquina $ exit
usuario@maquina $
```

La diferencia entre `su usuario` y `su - usuario` es, que mientras que con el primer comando simplemente adquirimos los permisos de *usuario*, con el segundo comando es como si estuviésemos haciendo login desde el principio con *usuario*, así, todas las variables de entorno y demás serán cargadas igual que si hubiésemos hecho login realmente. Esto también se aplica para root (`su -`). La shell que se arranca mediante la segunda forma se llama *shell de login*, y como puedes comprobar, no se puede suspender como hicimos anteriormente. Salimos de ellas también con `exit`.

El usuario `root` puede usar `su` o bien `su -` sin necesidad de introducir ningún password para adquirir en un shell los permisos de cualquier usuario del sistema.

SUID

Existe un atributo especial en los permisos para que los archivos ejecutables puedan ser ejecutados con los permisos de su dueño, independientemente de quién sea el usuario que lo ejecute. Esto es considerado por algunos como una bondad de los sistemas UNIX, mientras que por otros se considera uno de sus pocos fallos. Al margen de ese debate, a nosotros, como usuarios normales, nos otorga una cierta comodidad de cara a aplicaciones como las que acceden directamente al grabador de CDs por ejemplo, que necesitan muchas veces permisos de root.

Así, en este ejemplo de las aplicaciones que graban CDs, asignaríamos su dueño a root. Poniendo el atributo SUID a estas aplicaciones, cualquier usuario que las ejecutase, lo haría con permisos de root (ya que en este caso sería el dueño del archivo ejecutable).

En casos como este, aplicar los atributos SUID a algunos programas resulta útil. Pero cuidado, aunque esto es una posibilidad en Linux, es algo que debe evitarse siempre que sea posible. Es muy peligroso desde el punto de vista de la seguridad del sistema. A la hora de ejecutar un usuario normal una aplicación que tenga el atributo SUID establecido, ejecuta ese proceso virtualmente como root. Si todo va bien, no tienen por qué existir problemas. Pero si la aplicación tuviese un fallo durante su ejecución, y retornase inesperadamente al shell por ejemplo, ¡el usuario normal habría adquirido permisos de root! sin necesidad de conocer su contraseña, y con esto podrá causar daños en el sistema y leer nuestra información confidencial.

Cuando hacemos un `ls -l` en un directorio donde exista un programa que sea SUID, en la columna donde se muestran los permisos, en vez de tener una `x` representando los permisos de ejecución, tendría una `s` indicando que ese programa es SUID, y que para los apartados (propietario, grupo propietario o resto de usuarios) donde esté esa "s", será posible ejecutarlo con los permisos del usuario propietario.

La forma de asignar el atributo SUID a los archivos ejectuables es la siguiente:

```
# chmod +s /usr/bin/miprograma
```

Recuerda que es mejor eludir este método por razones de seguridad.

sudo

`sudo` (SUperuser DO) es una herramieta que permite otorgar a un usuario o grupos de usuarios normales, permisos para ejecutar algunos comandos como root (o como otros usuarios) sin necesidad de conocer su password. Es posible que no esté instalado en tu distribución de Linux y tengas que instalarlo tú. En capítulos posteriores tienes información sobre cómo instalar software adicional en el sistema.

El fundamento de sudo reside en su fichero de configuración, el fichero `/etc/sudoers`. Este fichero tiene, en los casos más sencillos, dos partes: una parte de *alias* y otra parte de *reglas*. La parte de alias, lo que hace es "agrupar" de alguna manera listas de usuarios y listas de aplicaciones (incluso listas de máquinas de una red, pero esto es más específico y no lo explicaremos aquí). La parte de reglas define qué grupos de usuarios pueden usar qué grupos de programas con permisos distintos de los suyos y en qué condiciones pueden hacerlo. Un fichero sencillo que nos podría servir podría ser como:

```
#
# Parte de alias
#

Cmnd_Alias GRABAR = /usr/bin/cdrecord, /usr/bin/cdrdao
Cmnd_Alias APAGAR = /sbin/halt, /sbin/reboot

User_Alias USERSGRAB = usuario1, usuario2

#
# Parte de reglas
#

USERSGRAB ALL = NOPASSWD: GRABAR

%cdrom  ALL = NOPASSWD: GRABAR
%apagar  ALL = NOPASSWD: APAGAR
```

Esto con respecto al fichero `/etc/sudoers`. En la parte de alias, `Cmnd_Alias` indica una lista de comandos (programas) que serán luego referidos mediante el nombre que le demos (asginar alias aquí tiene similitud con asignar variables de entorno en el shell). `User_Alias` agrupa a una lista de usuarios bajo un mismo nombre (en nuestro caso USERSGRAB).

En cuanto a la parte de reglas, primero se especifican los usuarios (puede ser un alias definido anteriormente como en el primer caso, o bien puede ser un grupo de usuarios del sistema, precediendo su nombre por `%`), el `ALL` que sigue hace referencia a en qué máquinas podrán hacer esto, y el `NOPASSWD:` indica que lo harán con permisos de root y sin necesidad de teclear su password. Después viene el alias con los comandos que podrán ejecutar en las condiciones que hemos dado.

Recuerda que el fichero `/etc/sudoers` se edita con el comando **`visudo`**, por razones de seguridad, y como root. `sudo` no altera la estructura de permisos del sistema de ficheros de Linux, es decir, por muchos cambios que hagamos en el fichero de configuración de sudo, los permisos de los programas seguirán siendo los mismos. La diferencia está en que estos "permisos especiales" que estamos

otorgando a algunos usuarios se aplican cuando el programa que se quiere ejecutar se llama mediante sudo; así, un usuario que quiera ejectar el programa `cdrdao` con estos permisos especiales deberá hacerlo así:

```
$ sudo cdrdao [opciones]
```

Esto es lo más básico que necesitas saber sobre sudo para ejecutar algunos comandos cómodamente como usuario normal al tiempo que mantienes la seguridad del sistema. `sudo` es una herramienta que permite configuraciones mucho más complejas que las que hemos visto aquí; siempre puedes leer sus páginas del manual del sistema (`man sudo` y `man sudoers`), o visitar su página web (http://www.courtesan.com/sudo/).

Capítulo 6. Entrada y salida

En este capítulo vamos a ver cómo manejar la entrada y la salida de los datos desde la terminal BASH, a nivel más o menos básico.

Introducción a los conceptos de entrada y salida

Es posible que ya hayas visto en algún sitio los terminos *entrada* y *salida* aplicados al mundo de la informática (quizás como I/O -input/output- o bien E/S -entrada/salida-).

En realidad es algo más simple de lo que pueda parecer, como vamos a ver mediante ejemplos sencillos. Cuando estamos dentro del sistema usando la terminal bash, generalmente tecleamos comandos para que sean ejecutados. El hecho de que estemos simplemente presionando el teclado implica que estamos *entrando* datos al sistema mediante el teclado. Cuando en ejemplos anteriores hemos usado el comando **`ls`** , ha aparecido en pantalla un listado de archivos y directorios, claramente el sistema ha producido *salida* de datos mediante el monitor.

Hemos puesto dos ejemplos, el teclado como vía para entrar datos al sistema (o bien a un programa, como ya vimos con vim), y el monitor como vía de salida de los datos. Pero no son, ni mucho menos, las únicas posibilidades. Los distintos puertos físicos del ordenador pueden ser vía de entrada o de salida de datos, igualmente con la tarjeta de sonido, módem, tarjeta de red...

Aunque lo que nos interesa en esta sección, es manejar la entrada y la salida de datos de y entre ciertos programas. Preguntas como... ¿y qué puedo hacer con el listado que me da el comando **`ls`** además de mostrarlo por pantalla? o ¿qué otras formas de pasar datos y opciones a los programas existen además del teclado? A todo esto y a algunos detalles más daremos respuesta en este capítulo.

Los sistemas UNIX (y por ello también los Linux), son especialmente conocidos por su facilidad y eficiencia para pasar datos entre programas y programas o entre programas y dispositivos, siendo esta una de sus múltiples ventajas.

Otros conceptos importantes dentro de estos son los de *entrada estándar* y *salida estándar* (stdin y stdout respectivamente). Por defecto, stdin apunta al teclado (los datos, se toman generalmente por defecto del teclado), y stdout apunta a la pantalla (por defecto los programas suelen enviar la salida por pantalla).

Comandos principales asociados con la salida

Son, principalmente **`echo`**, **`cat`**.

En primer lugar, echemos un vistazo al comando **`echo`**. Este comando nos permite "imprimir" cosas en la pantalla.

Veamos algunos ejemplos:

```
usuario@maquina ~ $ echo "Hola, usuario de Linux"
Hola, usuario de Linux
```

Como ves, por defecto el comando **`echo`** recibe las cadenas de texto entre comillas (simples o dobles), y saca por defecto a la pantalla lo que le pasamos como argumento; aunque puede enviar la salida a otros sitios, como podrás comprobar más adelante.

Adelantándonos un poco, el shell puede hacer operaciones aritméticas básicas, y podemos aprovecharlas con el comando **`echo`**:

```
usuario@maquina ~ $ echo $[5*3]
15
```

Cambiemos de comando, vamos a aprender ahora el comando **`cat`**. Es un comando útil que será nuestro amigo a partir de ahora.

`cat` tiene asociada la entrada al teclado por defecto, y la salida a la pantalla. Asi que si lo llamamos sin argumentos, simplemente tecleando **`cat`**, tecleamos y pulsamos INTRO, volverá a imprimir en pantalla cada línea que nosotros tecleemos:

```
usuario@maquina ~ $ cat
Escribo esto
Escribo esto
Y escribo esto otro
Y escribo esto otro
```

Continuará hasta que le mandemos detenerse manteniendo pulsada la tecla CONTROL y presionando a la vez la tecla C. Puede parecernos poco útil el funcionamiento por defecto del comando **`cat`**. Pero observemos ahora el funcionamiento más común de este comando, que es mostrar el contenido de un fichero:

```
usuario@maquina ~ $ cat directorio/mifichero
Esto es el contenido del fichero
mifichero que se encuentra en ~/directorio.
```

Este es el uso más frecuente para el comando **`cat`**, tomar como entrada un fichero y sacar su contenido por la salida. Por defecto es por la pantalla, pero descubriremos después que puede enviar el contenido de un fichero por otras vías.

Como ya habrás imaginado, los ficheros de disco pueden ser objeto de entrada (leer su contenido) o de salida (escribir la salida de un determinado comando a un fichero). El comando siguiente, **`touch`** nos da la posibilidad de crear un fichero vacío con el nombre que nosotros le especifiquemos:

```
usuario@maquina ~ $ touch minuevofichero
```

Así crearemos un fichero nuevo vacío al que podremos posteriormente enviar la salida de nuestros comandos. Si listas el fichero con **`ls -l minuevofichero`** verás como efectivamente su tamaño es cero. No obstante, **`touch`** puede hacer alguna otra cosa, como explicaremos en secciones posteriores.

Otro comando útil es el comando **`grep`**. Este comando por defecto lee de stdin (la entrada estándar), y envía los datos a la salida estándar (stdout). El objetivo primordial del comando grep es, "devolver" las líneas que contienen la cadena de texto que le pasamos al principio. Por ejemplo:

```
$ grep hola
Esta frase no contiene la palabra, por lo tanto no la repetirá
Esta frase contiene la palabra hola, por lo que la repetirá
Esta frase contiene la palabra hola, por lo que la repetirá
```

Como ves, el comando **`grep`** ha identificado las líneas que contenian la palabra que le pasamos como primer argumento.

Pipes o *tuberías*

Hasta ahora hemos estado practicando de dónde toman los datos algunas de las utilidades más comunes, y hacia dónde envían su salida. Lo siguiente que tenemos que plantearnos sería... ¿no podríamos "conectar" de alguna forma la salida de un programa y hacer que otro programa lea de ella?

Efectivamente, podemos conseguir esto. El caracter **`|`** nos permite crear estas conexiones entre programas en el shell. Ejemplo:

```
$ cat mifichero | grep gato
```

Lo que este comando hará será lo siguiente: La primera parte antes de la barra vertical, conseguirá el contenido del fichero `mifichero`, y con la barra vertical, en lugar de sacarlo por la pantalla, lo enviará a la entrada estándar (stdin) de tal forma que el programa **`grep`** leerá desde stdin el contenido del fichero `mifichero`. El concepto del efecto que esto produce es fácil de entender: el comando que va detrás de la barra lee de la salida que dió el comando anterior. Es obvio que lo que obtendremos serán las líneas del fichero `mifichero` que contienen la palabra "gato" por la pantalla (stdout).

Como ves, no estamos pasando ninguna opción ni al comando `cat` ni al comando `grep`. Esto es porque el primero (cat) envía la salida por defecto a stdout, y grep, si recuerdas, lee por defecto de stdin (que al usar la barra vertical, hemos hecho que stdin sea la salida del primer comando y no las líneas que tecleemos a continuación como hicimos anteriormente). Pero los pipes con la barra vertical podremos emplearlos con otros programas que no tengan este comportamiento por defecto, esto es, que su salida no apunte por defecto a stdout (podría apuntar, por ejemplo, a un fichero) o que no lean por defecto de stdin. Podremos hacer que tengan este comportamiento pasándoles determinadas opciones si disponen de ellas. Es muy común la opción **`-`** (un solo guión) para hacer que un programa lea de stdin o escriba a stdout si tiene esta funcionalidad; seguro que te encuentras algun comando que tenga esta posibilidad más adelante.

Otros caracteres que nos permiten crear "pipes", son **`<`** y **`>`**. Generalmente, este par de caracter trabajan con un fichero a un lado y un comando a otro. Me explico; veamos primero el caracter **`<`**:

```
$ grep gato < mifichero
```

Esta línea es equivalente a la del ejemplo anterior. El contenido del fichero que ponemos a la derecha del signo se va a la entrada estándar de la que lee el comando que ponemos a su izquierda, por lo tanto esta línea conseguiría exactamente el mismo efecto que la del ejemplo anterior.

Ahora el signo contrario. A la izquierda del signo **`>`** ponemos un comando, y a la derecha el nombre del fichero al que queremos que se escriba la salida del comando. Por ejemplo:

```
$ ls -la > listado
```

La lista del contenido del directorio actual que el comando `ls` sacaría normalmente por pantalla, en vez de eso se ha escrito en el fichero `listado`. Si el fichero no existía, se ha creado, y si existía y tenía contenido, ha sido sobreescrito, asi que tengamos cuidado.

El uso de los signos **`>>`** juntos, tiene una funcionalidad similar al uso de un signo solamente, y puede sernos útil muchas veces:

```
$ echo "Esta es otra línea" >> texto
```

Vemos en qué es similar a lo anterior; la línea "Esta es otra línea se ha escrito al fichero `texto`. Si el fichero no existía, ha sido creado, y la diferencia, si existía y tenía contenido, la línea se ha escrito al final del fichero, debajo del contenido que ya tenía. Este comportamiento nos puede ser muy útil de cara a ir añadiendo líneas a un archivo sin necesidad de tener que abrirlo con un editor.

Otras utilidades y detalles de la E/S en el shell

Aunque es adelantar un poquito, vamos a ver cómo la entrada estándar, la salida estándar y alguna cosa más "están" en el sistema de archivos.

La entrada estándar (stdin) está enlazada mediante el dispositivo (sobre lo que se hablará más adelante) `/dev/stdin`:

```
$ echo "hola" | cat /dev/stdin
hola
```

Con el comando `echo`, escribimos la palabra "hola" a la salida estándar, que con la barra vertical, se pasa a la entrada estándar, pero como le estamos pasando un argumento a `cat`, lee de `/dev/stdin` que... ¡sorpresa! ha resultado ser la entrada estándar. Si hubiesemos escrito el mismo comando quitando `/dev/stdin`, el resultado habría sido el mismo, puesto que si a `cat` no le pasamos ningún argumento, lee desde stdin. No te preocupes si no puedes comprender del todo lo que hace este ejemplo, simplemente tú asocia `/dev/stdin` con la entrada estándar que es accesible desde el sistema de ficheros.

La salida estándar está asociada a `/dev/stdout`:

```
$ cat fichero > /dev/stdout
--contenido de fichero--
```

Aquí, estamos enviando el contenido de `fichero` a `/dev/stdout` que, curiosamente, igual que en el caso anterior, ha resultado "conectar" con la salida estándar, esto es, la pantalla.

Por último, os presentaremos a nuestro basurero particular :-) , que es `/dev/null`. Nos puede ser útil para cuando queramos despreciar la salida de un comando, que ni salga por pantalla ni que se quede escrito en un fichero, simplemente que se pierda:

```
$ cat mimayorfichero > /dev/null
```

Una ventaja de /dev/null con respecto a los basureros físicos es que nunca se llena ;-) , podemos "arrojarle" toda salida que queramos despreciar.

more y less

`more` y **`less`** son dos comandos útiles a la hora de paginar. Paginar es ver un archivo o una salida larga por partes (o páginas), puesto que de otra forma sólo podríamos ver el final de esta salida larga. Pongamos por ejemplo que quisiésemos leer un archivo largo. Como ya sabes, esto podrías hacerlo con el editor vim, pero también podrías hacerlo con cat y la ayuda de estos dos comandos:

```
$ cat archivolargo | more
$ more < archivolargo
$ more archivolargo
```

Como ya sabemos, los dos primeros comandos son exactamente equivalentes. Pero el tercero tiene una pequeña diferencia con respecto a los anteriores. Si introducimos uno de los dos primeros con un archivo largo, entonces veremos que el programa more nos indica en la parte inferior "Más" o "More", y nada más; mientras que si introducimos el tercer comando, en la parte inferior de la terminal nos aparecerá "More (15%)", esto es, el porcentaje del fichero que estamos mostrando. La diferencia pues entre estos comandos es que en los dos primeros, more no conoce la longitud de los datos que vienen por la stdin, simplemente los procesa conforme el usuario se lo pide; mientras que en el tercer caso, el funcionamiento de more es conocer la longitud del fichero que se le pasa como primer argumento, y después paginarlo igual que hacía con los datos de stdin. Esto es importante; en UNIX y en Linux podemos trabajar con la stdin y stdout conforme vayamos necesitando sus datos, sin necesidad de saber cuál es su longitud o sin miedo de que estos vayan a perderse.

En lo que respecta al comando **`more`**, podemos seguir paginando hacia abajo con la barra espaciadora, hasta llegar al final del fichero o de los datos de entrada; o bien presionar la tecla **q** en cualquier momento para abandonar el programa.

El comando **`less`** es un "more" de GNU mejorado. Si recuerdas, con more sólo podíamos paginar hacia abajo, y bajar página por página. Con less podemos hacer algunas cosas más, como volver arriba o abajo, o bajar o subir línea por línea. Se puede usar igual que los ejemplos anteriores del comando more, con la diferencia de que una vez que estemos viendo la entrada paginada, además de poder bajar con la barra espaciadora, podemos subir y bajar con las flechas de dirección, o bien con las teclas de "avanzar página" y "retroceder página" de nuestro teclado. En el momento de salir, lo haremos igualmente presionando la tecla **q**.

Comandos útiles de less

Frente a **`more`**, **`less`** nos ofrece algunos comandos que nos ayudan a buscar cadenas y patrones dentro del buffer que less esté visualizando. Como ya es costumbre en este manual os proporcionamos una tabla con los comandos más utilizados en less, por supuesto hay muchos más, pero la idea no es reproducir la página del manual.

Para invocar a **`less`** podemos hacerlo con un *pipe* o directamente así:

```
$ less fichero1 fichero2 fichero3
```

Tabla 6-1. Órdenes más comunes de less

Comando	Descripción	Ejemplo de uso si aplica
:n	Examina el siguiente fichero en la lista de argumentos de less	--
:p	Examina el fichero anterior en la lista de argumentos de less	--
:e fichero	Examina el *fichero* pasado como argumento	:e /etc/xml/docbook/catalog

Comando	Descripción	Ejemplo de uso si aplica
:x N	Examina el primer fichero de la lista si N no ha sido especificado, en caso contrario examina el fichero que esté en la posición N de la lista.	:x 6
:d	Elimina el fichero que se está examinando de la lista	--
= o ^G o ^F	Da información adicional sobre el fichero incluyendo el nombre, el número de línea el %...	=
!comando	Nos permite ejecutar el comando como si estuvieramos en una shell. !! ejecutará el último comando ejecutado. Si se incluye en el comando el símbolo % será reemplazado por el fichero que se está examinando en ese momento, y # por el ultimo fichero examinado. Si no se especifica un comando devolverá una shell que se tomará de la variable de entorno SHELL.	!xmllint --valid --noout %
/patrón	Se busca en el fichero la cadena especificada por patrón. Si no se especifica ningun patrón se utilizará el último patrón buscado. Si el primer caracter de patrón es un * se buscará el patrón en todos los ficheros de la lista. La búsqueda es siempre hacia posiciones posteriores al cursor, es decir, hacia adelante.	/*gfdl
?patrón	Idéntico en todo a / pero la búsqueda se hace hacia atrás.	?fdisk
s fichero	Si el buffer que se está visualizando no es un fichero sino información leida de un pipe podemos guardar el buffer en *fichero*	s ~/temp/log

Nota: Una letra precedida de ^ significa que hay que pulsar primero la tecla **Ctrl** de forma que ^*G* es lo mismo que decir: **Ctrl**-**G**

Nota: Es muy util ajustar la variable de entorno PAGER a less (export PAGER=/usr/bin/less) dado que así las páginas del manual se mostrarán con `less` y podrás utilizar sus funciones de

búsqueda de patrones.

Recuerda que puedes utilizar more y less con otros comandos distintos que produzcan mucha salida, y usarlos para ver la salida detalladamente.

Otro pequeño truco, es que si estás en una terminal, puedes ver las líneas anteriores (que ya no se muestran en pantalla) manteniendo pulsada la tecla de las mayúsculas (no el bloqueo de mayúsculas) y a continuación presionando las teclas "avanzar página" y "retrocer página" de tu teclado para moverte por líneas anteriores de tu sesión en una terminal.

stderr y otras redirecciones

Hemos visto stdin y stdout, entrada y salida estándar, respectivamente. Sabemos que el siguiente ejemplo listará los contenidos del directorio actual y despreciará ese listado, enviándolo a `/dev/null`:

```
$ ls ./ > /dev/null
$ ls directorio_inexistente/ > /dev/null
ls: directorio_inexistente: No existe el fichero o el directorio.
```

En el primer caso, todo fue como esperábamos, la salida del listado fue simplemente despreciada. Pero... ¿qué ha pasado en el segundo caso? Con el signo > le decíamos que enviase la salida estándar, en este caso, a /dev/null. Y, como no te hemos mentido, la salida estándar fue a /dev/null. Lo que ha pasado es que el mensaje de error no ha ido a stdout; porque de haber sido así habría sido despreciado. ¿Dónde ha ido entonces el mensaje de error? Es sencillo, a **`stderr`**. La mayoría de los programas imprimen stderr por pantalla; aunque podrían (como hacen algunos), escribir los mensajes de error a un fichero, o por qué no, despreciarlos.

Algunos programas hacen estas cosas con stderr; cosas que ya hemos aprendido nosotros a hacer con stdout. Una solución posible sería que stderr fuese a stdout, y a partir de ahí, nosotros ya pudiésemos redirigir stdout como sabemos. Conseguir esto no es difícil, se hace con el operador **`>&`** :

```
$ ls directorio_inexistente/ > /dev/null 2>&1
```

Ahora sí será despreciada stderr; pero analicemos este comando por partes, y entendámoslo bien. Debemos saber que en él, "2" representa a stderr, y que "1" representa a stdout. Más cosas; este comando tiene dos partes. La primera es `ls directorio_inexistente/ > /dev/null` y la segunda es `2>&1`. En las redirecciones de salida complejas, como es esta, el shell las interpreta "al revés", esto es, de derecha a izquierda. Primero interpreta lo que hemos dicho que es el segundo comando, y luego interpreta el primero. Lo que ocurre entonces es lo siguiente: primero, stderr va a stdout por el segundo comando que se procesa en primer lugar, y luego stdout (que ya "contiene" a stderr) se va a /dev/null. Entonces conseguimos el efecto deseado, despreciar tanto stdout como stderr; aunque podríamos haber hecho otras cosas con ellas, lo importante es que ambas "salidas" habrían ido al mismo sitio.

Pero puede ocurrirte (y de hecho te ocurrirá), que desees tratar por separado stdout y stderr. Si se entendió el ejemplo anterior esto no debería causar problemas. Vamos a poner, como caso práctico, que vamos a listar una serie de archivos o directorios; los que existan se escribirán sus detalles en un fichero, y los que no obtendremos el error en otro fichero distinto:

```
$ ls -l fichero1 fichero2 fichero3 > existentes 2>inexistentes
```

```
$ ls -l fichero1 fichero2 fichero3 1>existentes 2>inexistentes
```

Estos dos comandos son absolutamente equivalentes. Recordemos que este tipo de comandos son interpretados de derecha a izquierda. En el primero de ellos, al principio stderr se escribirá al fichero inexistentes, y después el resto (o sea, stdout) se escribirá al fichero existentes. En el segundo, primero 2 (stderr) se escribirá a inexistentes, y después 1 (stdout) se escribirá a existentes. Usando cualquiera de ellos, al final tendremos en el fichero `existentes` los detalles de los ficheros existentes y en el fichero `inexistentes` el error producido por los ficheros (o directorios) que no existen. Este ejemplo es bastante ilustrativo de cómo podemos tratar separadamente stderr y stdout.

Por el momento ya tienes suficiente entrada y salida. Verás las amplias posibilidades que te ofrecen estos conceptos más adelante trabajando con otros programas, o incluso, a la hora de crear los tuyos.

Capítulo 7. Introducción al shell scripting

Si ya has practicado un poco con el shell bash, te habrás dado cuenta de que se pueden hacer muchas cosas interesantes con él. Aunque hay veces, que para conseguir determinado resultado necesitas introducir, por ejemplo, 3 o 4 comandos; o tenemos que evaluar una condición manualmente (por ejemplo, con `ls` comprobamos manualmente si un archivo existe). Los *scripts de shell* nos dan la posibilidad de automatizar series de comandos y evaluciones de condiciones.

Un *script* es como se llama a un archivo (o conjunto de archivos) que contiene instrucciones (en nuestro caso comandos de bash), y que necesita de un programa ayudante para ejecutarse (en nuestro caso la propia terminal bash será el programa ayudante). Un script tiene cierta similitud con un programa, pero existen diferencias. Generalmente, los programas están en lenguaje de máquina (la máquina lo entiende directamente), mientras que los scripts son archivos que contienen en formato texto los comandos o instrucciones que la aplicación ayudante ejecutará.

Aunque en principio esto puede resultar un tanto abstracto, vamos a verlo despacio y con ejemplos. Si no has programado nunca, en este capítulo encontrarás el aprendizaje del shell-scripting en bash como una sencilla introducción a la programación, que te ayudará a comprender después los fundamentos de otros lenguajes de programación. No te preocupes, porque lo vamos a explicar todo de forma sencilla y no te quedará ninguna duda.

Nuestro primer script en bash

Ahora que has practicado con bash y con el editor vim, ya estás preparado para crear tu primer script bash. Si recordamos, un script era un archivo de texto plano con comandos; así que haz **`vim miscript.sh`** y pon en él lo siguiente (lo explicaremos detenidamente a continuación):

```
#!/bin/bash

# Esta línea será ignorada
# Esta también

echo "Hola"
echo "Soy un script de shell."
```

Como ya sabemos, la extensión en los archivos en Linux no tiene significado útil. Es por convenio general, por lo que a los scripts de shell se les llama con la extensión ".sh" (de SHell). De este modo identificaremos fácilmente nuestros scripts entre el resto de archivos de un directorio.

Observemos la primera línea: `#!/bin/bash`. Esta línea es un tanto especial, y es característica de todos los scripts en Linux, no solamente los de bash. Tras `#!` indicamos la ruta a la aplicación ayudante, la que interpretará los comandos del archivo. En nuestro caso es bash, así que ponemos ahí `/bin/bash`, que es la ruta hacia la aplicación `bash`. Como más adelante tendrás tiempo de descubrir, hay otros tipos de scripts (Perl, Python, PHP...) para los cuales deberemos indicar la aplicación ayudante correspondiente.

Como ves, no importa que pongamos líneas en blanco, pues serán ignoradas. Podemos ponerlas después de cada bloque de comandos que tengan cierta relación, para dar claridad y legibilidad al conjunto de comandos del script.

Las dos siguientes líneas comienzan por el carácter #. En los scripts de bash, las líneas que comienzan con este signo son ignoradas, y se llaman *comentarios*. Podemos usarlas para explicar qué hace el

grupo de comandos siguiente, de tal forma que cuando editemos el script en el futuro nos sea más fácil descubrir qué es lo que está haciendo en cada sitio; o podemos usarlos para dar cualquier otra información importante de cara a la edición del archivo. Veremos más ejemplos de la utilidad de los comentarios después.

Las dos últimas líneas son dos comandos, de los que luego serán ejecutados por bash. Como sabemos, el comando `echo` saca por stdout (por defecto por pantalla) lo que le pasemos como argumento, en este caso dos frases. Podemos añadir cualquier comando normal de los que hemos visto, que será ejecutado normalmente. Después escribiremos algunos scripts más complejos, para que te hagas una idea de para cuánto pueden servirnos.

Guardemos el script y salgamos de vim. (Más información: Capítulo 4). Ahora lo ejecutaremos. La primera forma de hacerlo es como sigue:

```
$ bash miscript.sh
Hola
Soy un script de shell
```

Como ves, todos los comandos del script han sido ejecutados. Aunque la forma más frecuente de ejecutar scripts es, darles primero permisos de ejecución, como ya sabemos del capítulo de usuarios, grupos y permisos:

```
$ chmod +x miscript.sh
$ ./miscript.sh
Hola
Soy un script de shell
```

Al darle permisos de ejecución, podemos ejecutar ahora nuestro script como si de un programa normal se tratase. Esto es posible gracias a la primera línea del script, que hace que se llame inmediatamente a la aplicación ayudante para procesar los comandos. Hemos podido observar que a nivel del shell, nuestro script es un programa como otro cualquiera, que podemos ejecutar e incluso ponerlo en algún directorio listado en la variable de entorno `$PATH`, como por ejemplo `/usr/local/bin` y ejecutarlo como si de un programa del sistema se tratase, simplemente tecleando su nombre. Creo que ya te vas imaginando la gran utilidad de los scripts de shell...

Variables

Recordamos que una variable es un nombre al que le hacemos corresponder un valor. Este valor puede ser un valor numérico o bien una cadena de texto indistintamente en los scripts de shell. Los nombres de las variables conviene que comiencen por una letra, después veremos por qué.

En el siguiente ejemplo aprenderemos a usar las variables en nuestros scripts:

```
#!/bin/bash

# Script de muestra del uso de las variables
#
# Asignación de una variable:
#
variable=5
otravariable=10
tercera=2
```

```
   resultado1=$[5*10/2]
   resultado2=$[$variable*$otravariable/$tercera]

15 echo " "
   echo "El resultado de 5*10/2 es $resultado1 ,"
   echo "que efectivamente coincide con $resultado2 "
   echo " "

20 frase="Introduzca lo que usted quiera:"
   echo $frase
   read entrada_del_usuario
   echo " "
   echo "Usted introdujo: $entrada_del_usuario"
25
   exit 0
```

Repasemos el ejemplo despacio. En primer lugar tenemos la línea que será siempre la primera en nuestros scripts, y que ya conocemos, la que dice cuál es el programa que ejecutará las instrucciones del script.

Siguiendo tenemos la *asignación* de algunas variables, esto es, estamos dando valor a algunas variables que luego usaremos. Observa que entre el nombre y el valor, NO HAY ESPACIOS, es variable=valor. Procura recordarlo. Ahora cada variable tiene el valor que le hemos asignado. Podríamos cambiarlo simplemente de la misma forma, pero poniendo otro valor distinto, que sobreescribiría a cualquiera que tuviese antes la variable. Así, **`tercera=7`** haría que el valor de la variable `tercera` fuese de ahí en adelante 7.

Después nos encontramos con el comando `$[ ]`, que nos permite hacer operaciones aritméticas, obteniendo el resultado. Después veremos más detenidamente el hecho de que un comando DEVUELVA un valor. De momento trata de comprender este ejemplo. En la primera de esas líneas, la variable `resultado1` se queda con el valor resultante de efectuar la operación entre los números. En la línea siguiente, vemos que para REFERIRNOS al valor que tiene una variable, colocamos delante de su nombre el signo `$`. Dondequiera que pongamos ese signo delante del nombre de una variable, todo ello será reemplazado por el valor de la variable. Así, la expresión que se evalúa al final es la misma que la de la línea anterior, pues todas las variables quedan sustituidas por sus valores reales antes de que se haga la operación. Como ves, es bastante sencillo.

En las líneas siguientes usamos el ya conocido comando `echo`. Es muy conveniente que en nuestros scripts pongamos entre comillas lo que `echo` debe sacar por pantalla. Así, en la primera línea tenemos una forma sencilla posible para generar una línea en blanco en la salida, como podremos apreciar cuando después ejecutemos el script. Como ves, dentro de las comillas del argumento de `echo`, podemos hacer que aparezca el valor de una variable procediendo como ya hemos aprendido. El comando `echo` será nuestro mejor aliado para hacer que nuestro script saque información por la pantalla.

A la variable `frase` le vamos a asignar una cadena de texto. Date cuenta de que esta vez tampoco hay espacios en medio de la asignación y que EL TEXTO VA SIEMPRE ENTRE COMILLAS, pudiendo ser simples o dobles. La estructura es entonces variable="texto". En la línea siguiente, con el comando `echo` de nuevo, podemos sacar por pantalla el texto que habíamos guardado en la variable.

Atención a la línea siguiente. El comando `read` va a ser nuestra vía de comunicación con el usuario en nuestros scripts. Funciona de este modo: primero, ponemos **`read`**, el comando que hace todo el trabajo, y a continuación, ponemos el nombre de la variable donde queremos que la entrada del usuario quede almacenada (en nuestro caso, la variable es `entrada_del_usuario`). Cuando a la

hora de ejecutarse, el script llegue a esa línea, esperará a que el usuario escriba algo y presione la tecla INTRO. En ese momento, lo que el usuario escribió antes de de presionar INTRO quedará guardado en la variable, y luego podremos hacer uso de estos datos que el usuario introdujo, como hacemos en la línea siguiente, sacarlos de nuevo por pantalla.

Por último tenemos el comando `exit`. Este comando al final del script no es especialmente útil, pero si pusiésemos cualquier cosa debajo de él ya no sería ejecutada. Lo usaremos después cuando queramos que "si se da tal condición, entonces que el programa (script) termine".

Para terminar de comprender este segundo script de ejemplo, ponlo en un fichero llamado, por ejemplo, `variables.sh` con el editor vim. Dale permisos de ejecución como hicimos antes y después ejecútalo:

```
$ ./variables.sh

El resultado de 5*10/2 es 25 ,
que efectivamente coincide con 25

Introduzca lo que usted quiera:
Yo tecleo esta frase

Usted introdujo: Yo tecleo esta frase.
```

Hacer scripts de shell no es difícil, pero tampoco se nace sabiendo escribirlos. Como habrás podido adivinar, la mejor forma de aprender a escribir scripts es... ¡escribiendo scripts! Así que deberías considerar crearte los tuyos propios con los conceptos que hemos aprendido y practicar así para consolidarlos.

Comandos posibles

Una duda que puede surgir es... ¿qué comandos puedo escribir en mis scripts? ¿Tengo alguna limitación?

De aquí otra de las bondades de los scripts en bash: todos los comandos que escribiremos en bash pueden escribirse en un script de bash; esto incluye cualquier programa, redirecciones de entrada o salida, pipes, etc. Usando comandos del sistema, podemos hacer que nuestros scripts hagan casi cualquier cosa que nosotros queramos, y que se comporten como si de complejísimos programas se trataran. Esto es, para borrar un fichero dentro de un script, simplemente necesitamos usar el comando **`rm`**, sin preocuparnos por cómo funciona **`rm`** internamente. Con los comandos que ahora conocemos, nuestros scripts podrán: listar ficheros o directorios, borrar, despreciar salidas, buscar en cadenas de texto, pedir datos al usuario e interpretarlos...

Lo mejor, a parte de conseguir una secuencia automatizada (no será necesario teclear todos los comandos que necesitemos para una determinada cosa todas las veces; simplemente los escribiremos en un script y lo ejecutaremos con una sola línea cada vez que lo necesitemos); es que podemos hacer que los argumentos que le pasemos a un comando en el script sean variables. Poco a poco irás comprendiendo lo ventajoso de que los argumentos de los programas que ejecutemos en nuestros scripts sean variables.

Hacer notar una cosa. Cuando una de nuestras variables sea el argumento de un programa, y si lo llamamos con muchos argumentos, es muy conveniente para evitar errores extraños, entrecomillar la variable:

```
MI_VARIABLE=/un/directorio

# No muy adecuado:

ls -l $MI_VARIABLE

# Bastante mejor:

ls -l "$MI_VARIABLE"
```

Cuidado, en este caso las comillas *deben* ser dobles.

Todavía hay muchos comandos que no conocemos, pero cuando aprendas algunos nuevos, recuerda que pueden servir para dar funcionalidades extra a tus scripts.

Concepto de *valor de retorno*

Cuando un programa termina de ejecutarse, *devuelve* o *retorna* un valor. En el caso de los programas, el valor de retorno es un número. El valor es cero (0) si el programa finalizó con éxito o distinto de cero si el programa no finalizó con éxito. Cada uno de los valores distintos de cero va asociado a una causa distinta (sabiendo este valor podríamos saber por qué razón falló el programa), pero puede que por ahora no lo necesites.

El valor de retorno es un concepto un poco abstracto, pero no es difícil de comprender. El valor de retorno no se imprime por la pantalla, es decir, cuando un programa termina no saca por pantalla un cero, pero nosotros podemos saber cuál es ese valor.

En el shell bash (y por extensión, en nuestros scripts) el valor de retorno del último programa o comando ejecutado queda almacenado en la variable especial **`$?`** .

Veamos un ejemplo:

```
#!/bin/bash

echo "Listando archivo existente..."
ls archivo_existente
echo "El valor de retorno fue: $?"

echo " "

echo "Listando archivo inexistente..."
ls archivo_inexistente
echo "El valor de retorno fue: $?"

exit
```

En el primer caso, pedimos a `ls` que liste un archivo existente. Lo lista con éxito y por tanto el valor de retorno vale cero. En el segundo caso le pedimos que liste un archivo inexistente. Como eso no es posible, el comando no termina con éxito y el valor de retorno es distinto de cero.

La utilidad de conocer el valor de retorno al ejecutar un programa en nuestros scripts salta a la vista: cuando veamos las sentencias condicionales podremos hacer que el script haga una cosa u otra en función de si un programa terminó con éxito o no.

Volveremos a ver el valor de retorno cuando estudiemos las funciones.

Pasar y procesar argumentos

Los comandos que ejecutamos normalmente pueden recibir opciones y argumentos (parámetros), como por ejemplo **`ls -l ./`** . En ese ejemplo, ls es el comando, -l es una opción y ./ es un argumento. En esta sección veremos las variables especiales que nos permitirán luego hacer que nuestros scripts se comporten del mismo modo, que puedan recoger opciones y argumentos.

```
#!/bin/bash

echo "\$0 contiene $0"
echo "\$1 contiene $1"
echo "\$2 contiene $2"

echo "En total hay $# parametros"
```

Llama a este script `args.sh` y dale permisos de ejecución. Antes de ejecturalo vamos a explicarlo.

En primer lugar tenemos la barra invertida \ . Usaremos la barra invertida para *escapar* caracteres especiales. Si queremos sacar por pantalla con el comando `echo` unas comillas o un signo del dolar, la única forma es anteponiéndole una barra invertida, de otro modo sería interpretado como comillas de fin de cadena de texto o como variable respectivamente.

Luego tenemos las variables `$N`, donde N es un número natural: 0, 1, 2, etc. `$0` contiene el comando ejecutado, `$1` el primer parámetro, `$2` el segundo parámetro, y así sucesivamente.

Por último tenemos la variable especial `$#` . Esta variable contiene un número entero, que vale el número de parámetros que pasamos al script sin contar `$0`. Ejecutemos el script de dos formas distintas para comprender todo esto:

```
$ ./args.sh
$0 contiene ./args.sh
$1 contiene
$2 contiene
En total hay 0 parametros
```

```
$ ./args.sh buenos dias
$0 contiene ./args.sh
$1 contiene buenos
$2 contiene dias
En total hay 2 parametros
```

Estas dos ejecuciones han debido servir para ilustrar cómo funcionan estas variables especiales para recibir parámetros en nuestros scripts. Conforme sigamos aprendiendo veremos cómo podemos hacer que nuestro script se comporte de una forma u otra dependiendo de los parámetros que se le pasen. Cada vez más, nuestros scripts se irán pareciendo a programas "de los buenos".

Todavía hay más... puede que no se te ocurra cómo listar todos los parámetros que se le pasen al script, sean cuantos sean. **`shift`** nos da la solución. Hay algunas cosas que debemos saber antes de poder usar `shift` para listar los parámetros pasados a un script. No obstante, hay un ejemplo en la sección de nombre *Funciones*, que aplica tanto a funciones como al script en sí mismo.

> **Variable de proceso:** Aunque todavía no hemos visto los procesos, debemos hacer notar que existe la variable especial `$!` . Esta variable contiene el ID de proceso del último comando ejecutado. Todo esto será detallado en el capítulo de procesos.

Evaluación de condiciones

Muchas veces nos será necesario hacer que nuestro script se comporte de una forma u otra según el usuario introduzca uno u otro argumento, o según una variable tenga tal o cual valor. En esta sección pasamos a ver los comandos y estructuras que nos permiten hacer esto.

El comando que nos permite evaluar condiciones en bash es **`test`**. El *valor de retorno* de `test` será cero (0) si la *expresión* es VERDADERA (TRUE), y en cambio, será uno (1) si la expresión que le pasamos a `test` es FALSA (FALSE).

Lo que queremos evaluar recibe el nombre de EXPRESIÓN. `test` recibe las expresiones de un modo un tanto especial. En el siguiente ejemplo comprobamos si dos números son iguales o no:

```
$ test 7 -eq 7
$ echo $?
0
$ test 7 -eq 8
$ echo $?
1
```

Primero, `7 -eq 7` es lo que se conoce como expresión. De esta forma tan peculiar al principio le decimos a `test` que compruebe si 7 es igual a 7 (-eq, abreviatura de 'equals', 'es igual a'). Así que ese primer comando quiere decir: ¿Es 7 igual a 7? Como ves, `test` no ha producido ninguna salida por la pantalla, sin embargo, si comprobamos su valor de retorno, vemos que es cero (0), lo cual indica que la expresión es verdadera, 7 es igual a 7, efectivamente.

En el siguiente comando estamos comprobando si 7 es igual a 8. Como era de esperar, el valor de retorno de `test` es uno (1), lo cual indica que la expresión es falsa; 7 no es igual a 8.

Ahora supón que antes de todos aquellos comandos hacemos:

```
$ a=7
$ b=8
$ test $a -eq $a
# Aqui seguiriamos con las dos variables
```

y que repetimos todo lo anterior, pero en vez de usar 7 y 8, usamos las variables $a y $b. No hay cambios, `test` reemplaza las variables por su valor y después efectúa la comprobación, siendo el resultado final el mismo.

> **Sugerencia:** Para no tener que aprender de memoria la forma en que hay que introducirle las expresiones a `test`, puedes consultar las páginas del manual, aunque todavía no lo hemos descrito: **`man test`**
>
> Puedes moverte con las flechas de dirección y volver al shell pulsando la tecla `q`.

Entonces, para ver cómo y qué otras cosas se pueden evaluar con `test`, es muy recomendable seguir el consejo arriba expuesto.

Existe además otra forma de llamar a `test` que no es con el comando, y que en nuestros scripts puede ayudarnos a que con un sólo golpe de vista nos demos cuenta de qué es la expresión que se evalua. Esta otra forma consiste en poner entre corchetes (y con espacio) la expresión a evaluar:

```
$ [ 3 -gt 2 ]
$ echo $?
0
```

Estructuras condicionales con `if`

Hemos visto cómo evaluar condiciones con `test`. Con la ayuda de las estructuras condicionales que crearemos con `if` y lo que ya hemos aprendido, podremos por fin hacer que nuestros scripts se comporten de una forma u otra según las condiciones que nosotros especifiquemos.

Las estructuras condicionales tienen este aspecto:

```
if EXPRESION1; then
 # Bloque 1
elif EXPRESION2; then
 # Bloque 2
else
 # Bloque 3
fi
```

Este es el modelo más general con todas las opciones posibles. Si el valor de retorno de EXPRESION1 es cero, las líneas de comandos de "Bloque 1" serán ejecutadas. El resto del bloque `if` será ignorado en ese caso.

Si el valor de retorno de EXPRESION1 no es cero, pero el de EXPRESION2 sí que es cero, entonces el "Bloque 1" no será ejecutado, mientras que el "Bloque 2" sí que será ejecutado. El "Bloque 3" será ignorado. Podemos poner tantos `elif`'s como necesitemos.

Si ni EXPRESION1 ni EXPRESION2 ni cualquier otra de otros `elif`'s que pongamos retornan cero, entonces el "Bloque 3" será ejecutado.

Usamos `fi` para terminar la estructura condicional.

Como habrás supuesto, las EXPRESIONes son comandos con `test`. Veamos un ejemplo sencillo:

```
#!/bin/bash

echo "Intrduzca un numero: "

read input
```

```
if [ $input -lt 5 ]; then
        echo "El numero era menor que 5"
elif [ $input -eq 5 ]; then
        echo "El numero era 5"
elif [ $input -gt 5 ]; then
        echo "El numero era mayor que 5"
else
        echo "No intrudujo un numero"
fi
```

En este sencillo ejemplo hacemos que el usuario introduzca un número que queda guardado en la variable $input. Dependiendo de si el número es menor, igual o mayor que 5, el script se comporta de una forma u otra (podemos añadir más líneas de comandos después de los `echo`'s que aparecen ahí). Si el usuario no introduce nada o introduce una cadena de texto, lo que hay tras `else` es lo que será ejecutado, pues ninguna de las condiciones anteriores se cumple.

Date cuenta que en las expresiones entre corchetes podrían haberse sustituidos estos corchetes por **`test`**.

De todos modos, los `elif` y los `else` son opcionales, podemos crear una estructura condicional sin ellos si no los necesitamos. Como muestra, el siguiente ejemplo:

```
#!/bin/bash

if [ $# -ne 2 ]; then
        echo "Necesito dos argumentos, el primero"
        echo "es el fichero donde debo buscar y"
        echo "el segundo es lo que quieres que"
        echo "busque."
        echo " "
        echo "Uso: $0 <fichero> <patron_busqueda>"
        echo " "

        exit
fi

FICHERO=$1
BUSQUEDA=$2

if [ ! -e $FICHERO ]; then
        echo "El fichero no existe"

        exit
fi

NUM_VECES=‘cat "$FICHERO" | grep --count "$BUSQUEDA"‘

if [ $NUM_VECES -eq 0 ]; then
        echo "El patron de busqueda \"$BUSQUEDA\" no fue encontrado"
        echo "en el fichero $FICHERO "
else
        echo "El patron de busqueda \"$BUSQUEDA\" fue encontrado"
        echo "en el fichero $FICHERO $NUM_VECES veces"
fi
```

Este ejemplo ya se va pareciendo a un programa de los buenos. Estudiémoslo en detalle:

En primer lugar comprobamos que nos han pasado dos argumentos: el fichero donde buscar y la cadena de búsqueda. Si no es así, le indicamos al usuario cómo debe ejecutar el script y salimos.

A continuación comprobamos que el fichero existe. Si no es así, advertimos convenientemente y salimos.

Ahora viene la línea que hace todo el trabajo: `NUM_VECES=‘cat "$FICHERO" | grep --count "$BUSQUEDA"‘` Al entrecomillar con las comillas inclinadas el comando, su salida, en vez de salir por pantalla, quedará guardada en la variable NUM_VECES. Así, mediante `cat` y el pipe, `grep` cuenta el número de veces que el valor de $BUSQUEDA está en el fichero, que nos queda guardado en $NUM_VECES.

Lo siguiente habla por sí solo.

Coloca esas líneas en un script y dale permisos de ejecución. A continuación, ejecútalo de las distintas formas posibles: sin argumentos, con un fichero inexistente, con distintos patrones de búsqueda, etc.

Dentro de un bloque de una estructura condicional puedes poner otra (`if`s anidados), y dentro de esa otra puedes poner otra y así sucesivamente. También podrás poner cuando las veamos, bucles y otras estructuras, así como condicionales dentro de bucles, etc. Sólo la práctica te dará la suficiente soltura como para poder hacer que tus scripts hagan auténticas virguerías. ¡La práctica hace al maestro!

Comprobar el valor de una variable con `case`

Escribiendo scripts, es muchas veces necesario comprobar si el valor de una variable coincide con alguno de los que nosotros estábamos esperando, y si así es, actuar de una forma u otra dependiendo de este valor.

Ni que decir tiene que eso es posible hacerlo con `if`, y `elseif`, pero existe otra forma más eficiente y cómoda de hacer esto: con la estructura `case`. Se usa como sigue:

```
#!/bin/bash

case $VARIABLE in
 valor1)
  # Bloque 1
  ;;
 valor2)
  # Bloque 2
  ;;
.
.
.
.
 *)
  # Ultimo Bloque
  ;;
esac
```

`valor1` y `valor2` son valores que nosotros esperamos que la variable pueda contener, y Bloque 1 y Bloque 2 son los bloques de comandos con lo que queremos que se haga en cada caso. Podemos poner tantos valores reconocibles como queramos. Los puntos son precisamente para indicar aquello. El `*` (asterisco) es un *comodín* exactamete igual que hacíamos moviendonos entre directorios en capítulos

anteriores, cualquier cosa que no haya coincidido en alguna de las de arriba coincidirá aquí. Esto es completamente opcional. No obstante, en el ejemplo siguiente te darás cuenta de su utilidad.

Los dos ";;" es necesario que se pongan al final de cada opción.

```
#!/bin/bash

# case_arg.sh

if [ $# -lt 1 ]; then
        echo "Error. Esperaba al menos un parametro"
        exit 1
fi

case $1 in

        --opcion1)

        echo "--opcion1 es una opcion reconocida"
        exit 0

        ;;

        --opcion2)

        echo "--opcion2 es una opcion posible"
 exit 0

        ;;

        linux)

        echo "\"linux\" es un argumento reconocido"
 exit 0

        ;;

        *)

        echo "Parametro no reconocido"
        exit 1

        ;;
esac
```

En este ejemplo se hace que las opciones tengan que ir precedidas de un doble guión en forma larga (la única que reconoceremos en este caso), de esta forma damos a nuestro script un aspecto y un comportamiento más "profesional". Los argumentos no se suelen hacer preceder de ningún signo; aunque todo esto puedes ponerlo como tú prefieras, lo importante es que aprecies cómo funciona este tipo de estructura condicional.

Con `exit` salimos del script, haciendo que el valor devuelto sea 0 (éxito) o 1 (error) según los parámetros fueran reconocidos o no.

Dentro de cada uno de los bloques de comandos se pueden poner estructuras condicionales con `if`, o incluso otro `case`.

Bucles

Los bucles en bash-scripting difieren un poco de lo que son en la mayoría de los lenguajes de programación, pero vienen a cumplir una misión similar; a solucionar el problema de "Quiero que mi script haga esto mientras se de esta condición", o "Para cada uno de los elementos siguientes, quiero que mi script haga esto otro".

Así, tenemos dos tipos de bucles diferentes en bash: `for` y `while`.

Bucles con `for`

De los propósitos explicados anteriormente a solucionar por los bucles, `for` viene a solucionar el segundo. Veamos algunos ejemplos:

```
$ for i in pedro pepe juan jose; do
> echo "Hola $i"
> done
Hola pedro
Hola pepe
Hola juan
Hoja jose
```

En primer lugar apreciamos que todo lo que se puede hacer en un script también se puede hacer en bash. La estructura de `for` es la siguiente:

```
for nombre_var in LISTA ;  do
comando 1
comando 2
# Y asi los que queramos
done
```

`nombre_var` es el nombre de una variable (el que nosotros queramos, y sólo el nombre, no precedida de $). **`LISTA`** es una lista de elementos, que se la podemos dar como en el ejemplo anterior, ahí en ese comando, o podemos poner una variable que ya contenga una lista de elementos (en este caso sí que la precederíamos de $, para que sea reemplazada por los elementos de la lista que contiene). Después aprenderemos otras formas posibles de expresar esa lista.

Con este bucle lo que conseguimos es: **`comando 1`**, **`comando 2`**, etc. se ejecutan tantas veces como elementos haya en LISTA. Para la primera vez, `$nombre_var` valdrá el primer elemento de la lista, para la segunda vez que se ejecuten los comandos valdrá el segundo elemento de la lista, y asi sucesivamente hasta llegar al último elemento en el que el bucle habrá terminado (y el script seguirá su flujo de ejecución normal hacia abajo). Ahora ya deberíamos entender el ejemplo anterior.

`LISTA` puede ser una lista de archivos dada implícitamente. Más concretamente, podemos usar los *wildcards* que ya conocemos para referirnos a varios archivos con un nombre parecido:

```
#!/bin/bash

# Listador de archivos y directorios ocultos
5
DIR=$1

if [ -z $DIR ]; then
```

```
    echo "El primer argumento debe ser un directorio"
10  exit 1
   fi

   if [ ! -d $DIR ]; then
    echo "El directorio debe existir"
15  exit 1
   fi

   for i in $DIR/.*; do

20    echo "Encontré el elemento oculto $i"

   done

   echo "Saliendo..."
25
```

`.*` hace referencia a todos los archivos y directorios ocultos de `$DIR`. Al detectar `for` el wildcard, sabrá que los elementos de la lista son cada uno de los archivos coincidentes. Ejecuta el ejemplo anterior si quieres ver cómo funciona. Puedes hacer uso con este bucle de lo que ya sabes de wildcards referidos a archivos.

El elemento **`LISTA`** del bucle `for`, puede ser un comando entrecomillado con comillas laterales; un comando tal que su salida sea una lista de elementos. Así, sería lo mismo:

```
for i in *; do
```

que:

```
for i in ‘ls‘; do
```

Bucles con `while`

El otro tipo de bucle posible en bash-scripting es el bucle `while`. Este bucle viene a cumplir con el propósito de "quiero que se haga esto mientras que se dé esta condición". Su estructura es la siguiente:

```
while test CONDICION; do

 # Comandos a ejecutar
5
done
```

Como ya sabemos usar el bucle anterior, este lo entenderemos con algunos pocos ejemplos:

```
#!/bin/bash

j=0
5
```

```
while [ $j -lt 10 ]; do
 echo "j vale $j"
 j=$[$j+1]
done

# Aqui sigue el flujo de ejecucion normal del script
```

Este ejemplo es interesante. La condición para que el bucle se siga ejecutando es que j valga menos de 10. Observa la forma mediante la cual en cada ciclo del bucle sumamos 1 a la variable j.

Tanto en el bucle `for` como en el `while`, podemos usar **`break`** para indicar que queremos salir del bucle en cualquier momento:

```
#!/bin/bash

MINUM=8

while [ 1 ]; do
 echo "Introduzca un número: "
 read USER_NUM

 if [ $USER_NUM -lt $MINUM ]; then
  echo "El número intrododucido es menor que el mío"
  echo " "
 elif [ $USER_NUM -gt $MINUM ]; then
  echo "El número introducido es mayor que el mío"
  echo " "
 elif [ $USER_NUM -eq $MINUM ]; then
  echo "Acertaste: Mi número era $MINUM"
  break
 fi
done

# Comandos inferiores...

echo "El script salió del bucle. Terminando..."
exit
```

La condición es que el bucle se ejecute siempre (test 1 siempre retornará TRUE, y el bucle se seguirá ejecutando). Pero, si el número introducido es el mismo que el valor de $MINUM , `break` hará que se salga de el bucle y continúe ejecutando los comandos inferiores.

Ten en cuenta que en bash scripting, usando la notación `$[ ]` para hacer cálculos sólo podemos usar números enteros.

Además, dentro de un bucle `while` se puede utilizar cualquiera de los elementos ya conocidos, siempre que se haga correctamente.

Funciones

Si bien las funciones en bash scripting difieren de lo que son en la mayoría de los lenguajes de programación, sí comparten, en cambio, su razón de ser básica.

Una *función* en un script es un conjunto de líneas (comandos) a los que se les da un nombre especial y pueden ser ejecutados desde cualquier punto del script mediante el uso de ese nombre especial.

No se considera una buena práctica repetir en distintos sitios las mismas líneas de código. Así, se agrupan en una función, y LLAMANDO a la función por su nombre, esas líneas son ejecutadas dondequiera que la llamemos.

Una función se DEFINE de la siguiente forma:

```
function nombre_de_la_funcion () {

 ## Aquí van los comandos que se
 ## ejecutarán cuando llamemos a la
 ## función

}
```

`nombre_de_la_funcion` lo elegimos nosotros, y le podemos dar el nombre que queramos, con las mismas condiciones que el nombre de una variable (no espacios, no caracteres extraños, etc.). Entre las dos llaves (`{` y `}`) se introducen los comandos que se quiere que se ejecuten cuando la función sea llamada.

Un ejemplo sencillo podría ser el siguiente:

```
#!/bin/bash

# function1.sh

function uso () {
 echo "Este script recibe dos argumentos."
 echo "El primero debe ser --opc1 u --opc2 ,"
 echo "y el segundo debe ser un fichero existente."
 echo "--"
}

if [ $# -ne 2 ]; then
 uso
 exit 1
fi

case $1 in
 --opc1)
  if [ -e $2 ]; then
   echo "El script terminó con éxito"
   exit 0
  else
   uso
   exit 1
  fi
 ;;

 --opc2)
  if [ -e $2 ]; then
   echo "El script terminó con éxito"
   exit 0
```

```
     else
      uso
35    exit 1
     fi
    ;;

    *)
40   uso
     exit 1
    ;;
   esac
```

Se necesita definir la función antes de llamarla. Se llama a la función simplemente con el nombre que le dimos; ese nombre se convierte en un "comando posible" que lo que hace es ejecutar las líneas que hay dentro de la función.

De no haber usado aquí las funciones, tendríamos que haber reescrito los `echo` que dicen cómo se usa el script una y otra vez en cada sitio donde los hubiésemos necesitado. Mientras no se ejecute el script con los argumentos adecuados, llamamos a la función `uso` que explica al usuario cómo se debe llamar el script. Esta estrategia es muy frecuente y útil en scripts más complejos con muchas opciones.

Podemos definir tantas funciones como necesitemos, cada una con un nombre distinto. Además, al igual que los scripts desde la línea de comandos, las funciones pueden recibir parámetros, que se procesan de la misma forma: `$1 $2` y demás se corresponden con el primer y el segundo parámetro pasado a la función respectivamente.

Al convertirse la función en un "comando posible" dentro del script, los parámetros se le pasan igual que a cualquier otro comando:

```
   #!/bin/bash

   function lista_parametros () {
 5         echo "Se ha llamado a lista_parametros, con $#"
           echo "parámetros. Mostrando los parámetros:"
           echo " "

           b=7
10
           numparms=$#

           local j=1
           while [ $j -le $numparms ]; do
15                 echo "El parámetro $j contiene $1"
                   shift
                   local j=$[$j+1]
           done
   }
20
   lista_parametros a b c d dsfafd d

   echo $b

25 echo $j
```

La función `lista_parametros` en este ejemplo recibe varios parámetros del mismo modo que cualquier otro comando dentro del script, y se procesan igual que ya conocemos para los scripts.

Aquí hacemos uso de `shift` dentro de la función. Lo que shift hace es que $1 valga $2, que $2 valga $3, etc. para así recortar la lista de argumentos una posición quitando el primero de ellos. ¿Qué conseguimos? Pues es sencillo, listamos todos los parámetros posibles, puesto que imprimimos el primero por pantalla, luego el primero vale el segundo, que se imprime por pantalla, luego el primero vale el tercero, que se vuelve a sacar por pantalla, y así con tantos como tengamos. `shift` nos ofrece unas avanzadas posibilidades para procesar argumentos. En este ejemplo lo hemos visto dentro de una función, pero todo esto es también aplicable a un script en sí mismo, fuera de una función.

Para terminar con este ejemplo, nos fijamos en `local j=...`, esto significa que la variable `$j` sólo existe dentro de la función, de hecho, cuando hacemos `echo $j` fuera de la función, no obtenemos ningún valor. La variable `$b`, al no ser DECLARADA DE ÁMBITO LOCAL, sino GLOBAL, existe dentro de la función y fuera de ella, como se puede apreciar ejecutando el script. Cualquier otra variable con valor en el flujo principal del script existe dentro de las funciones que definamos.

Hemos dicho que cada función se convierte en una especie de "comando posible" dentro del script donde está definida. Como si de un comando se tratase, y al igual que hacíamos con `exit 0` o `exit 1` en el flujo principal del script, podemos hacer que la llamada a una función genere un valor de retorno de éxito o bien de error. Lo conseguimos con el comando **`return`**. Se usa exactamente igual que `exit` en el script, pero usaremos `return` dentro de las funciones.

Usar las funciones en nuestros scripts nos dá amplias posibilidades en el sentido de reducir el tamaño de éstos, y que sean más fáciles de interpretar si pasado un tiempo debemos estudiarlos y modificarlos. Además, otra ventaja de usar funciones para líneas que se repiten es que si no lo hiciésemos y quisiéramos modificar el comportamiento de esas líneas, deberíamos hacerlo en todos y cada uno de los sitios donde las hubiésemos puesto. Con las funciones, si modificamos las líneas de la función, ya queda modificada en todos los sitios.

Un ejemplo completo

Ahora vamos a ver en detalle un ejemplo de un script que usa muchas cosas de las aprendidas y además tiene un fin práctico: sirve para transpasar los archivos de una cámara digital a un directorio de nuestra máquina. Además, la función `echoc` llama al comando `echo` con los argumentos adecuados para que la salida sea con colores (esperamos comentar esto en otro capítulo). No te preocupes si no comprendes algunos comandos de los usados, pues son muy específicos; lo importante es ver que la correcta combinación de todo lo aprendido termina en scripts útiles como este:

```
#!/bin/bash
#
# Variables de configuración
#   + Acabado en / !!!!
DIRECTORIO_SALIDA=/home/camara/
#
#
# Funciones
#
function echoc()
{
        case $2 in
                "red")
   echo -n "^[[01;31m"
```

```
                        ;;
                "green")
    echo -n "^[[01;32m"
                        ;;
                "blue")
    echo -n "^[[01;34m"
                        ;;
        esac

        echo -n $1
        if test $3 = 1
        then
                echo -n "^[[00m"
        else
                echo "^[[00m"
        fi
}
#
#
function comp()
{
        echoc "[" "blue" "1"
        echo -n " "
        if test $1 = 0
        then
                echoc "OK" "green" "1"
        else
                echoc "!!" "red" "1"
        fi
        echo -n " "
        echoc "]" "blue" "0"
}


# Main
echo "Por favor, encienda la cámara y póngala en modo [|>]"

echo -n "Pulse una tecla para continuar"
read -n1
echo "."

echo -n "Montando la cámara... "
mount /camara 2>&1 > /dev/null
comp $?

echo "Copiando imágenes a /home/sergio/camara... "
for i in /camara/dcim/100nikon/*.jpg
do
        NUEVO_NOMBRE=‘basename $i | sed s:dscn:$DIRECTORIO_SALIDA:g‘
        echo -n "       $i -> $NUEVO_NOMBRE... "
        cp $i $NUEVO_NOMBRE
        comp $?
done

echo -n "Desmontando la cámara... "
umount /camara 2>&1 > /dev/null
comp $?
```

```
   if [ ‘/etc/init.d/samba status | awk ’{print $3}’‘ != "started" ]
75 then
           echo "Iniciamos samba... "
           /etc/init.d/samba start 2>&1 >/dev/null
           comp $?
   else
80         echo "Samba ya está iniciado."
   fi

   echo "Ya puede apagar la cámara con seguridad"
```

Conclusiones

Es importante que practiques a hacer tus propios scripts, aunque sean sencillos, para consolidar todo lo aprendido. Sólo con la práctica llegarás a dominar el bash shell scripting.

Recuerda la utilidad de los scripts a la hora de automatizar tareas que tengan que hacerse frecuentemente; es mejor hacer un script que teclear todos los comandos necesarios cada vez que lo necesitemos.

Posteriormente estudiaremos `cron`, que es un demonio que ejecuta lo que queramos a una hora determinada e incluso periódicamente. Nos será especialmente de ayuda crear scripts que, por ejemplo, hagan copias de seguridad automáticamente, y dárselos a `cron` para que los ejecute.

Otros comandos que iremos descubriendo nos permitirán hacer scripts que desempeñen tareas de lo más variado.

Capítulo 8. Instalación de Software adicional

Introducción

En este capítulo trataremos un ’problema’ al que cualquier usuario de Linux y de software libre tendrá que enfrentarse tarde o temprano.

Instalar software adicional en Linux es sencillo, sin embargo, dada la diversidad de distribuciones y sistemas de empaquetado del software las utilidades que manejan dichos paquetes son distintas.

Métodos de instalación

A la hora de instalar software adicional a la distribución que instalemos nos podemos encontrar con varios sistemas de paquetes. Existen, a grandes rasgos, cuatro sistemas de paquetes en todas las distribuciones de Linux: RPM, DEB, TGZ y EBUILD. Los tres primeros son binarios (ver la sección de nombre *Los paquetes binarios* en Capítulo 1), y el cuarto se trata de meta-paquetes (ver la sección de nombre *Meta-Paquetes* en Capítulo 1).

Los paquetes binarios contienen el software ya en código de máquina (a excepción de los Source RPM y demás que serán tratados más tarde), y pondrán los programas y ficheros de configuración en el sitio adecuado del árbol de directorios para que los otros paquetes puedan encontrarlos. Los sistemas de paquetes binarios se apoyan en una "base de datos" que guarda qué paquetes están instalados y cuáles no, la versión de estos, etc... Así, cuando instalamos un paquete binario, tal como un RPM o un DEB, además de crearse los ficheros necesarios para que el software pueda funcionar, se añade a esta base de datos una entrada diciendo que el paquete ha sido instalado y asimismo se guarda su número de versión. Algo parecido (pero no igual) ocurre con los ebuilds de Gentoo.

Conforme trates con estos sistemas de paquetes te podrá ocurrir que los datos de la base de datos de paquetes no coincidan con lo que realmente hay en la máquina. Esto puede ocurrir mediante el borrado accidental de ficheros sin desinstalar adecuadamente un paquete, o conflictos de versiones... En cualquier caso dispones de opciones que te permiten instalar o desinstalar paquetes incluso si la información de la base de datos de paquetes no es del todo coherente. Además, existen comandos para "reconstruir" o arreglar la base de datos de paquetes. Las páginas **man** de los programas que se mencionan a continuación te aportarán toda la información que necesitas al respecto.

Además de los paquetes de nuestra distribución disponemos de unos paquetes un poco especiales. Estos paquetes contienen el código fuente preparado para compilarse e instalarse en cualquier distribución. Por lo general no es difícil instalarlos pero puede dar bastante dolor de cabeza las primeras veces. Con estos paquetes no todo son penurias, también tienen sus ventajas. Entre las ventajas podemos destacar el control que adquirimos sobre la instalación del paquete y sus binarios, además de la posibilidad de hacer algunos cambios en el mismo si nos interesa (aplicando parches). Instalar software de este modo no implica que se guarde información en una base de datos de lo que se ha añadido al sistema como ocurre con los sistemas anteriores, por lo que si el paquete de código fuente no dispone de una opción para desinstalarlo, tendremos que borrar los ficheros a mano uno a uno en caso de querer quitar el software. Esto no es muy común ni necesario en la mayoría de los casos, porque aunque instalemos un software, mientras no lo estemos ejecutando "no nos molesta", el espacio ocupado en disco suele ser muy pequeño.

Escogiendo nuestro método (binarios vs. fuentes)

Como se ha nombrado antes, hay que decidir qué método vamos a utilizar. En principio siempre deberemos optar por los binarios de nuestra distribución a no ser que sepamos realmente lo que hacemos.

Compilar un programa desde las fuentes puede sernos útil en máquinas donde se necesite un rendimiento extremo o en casos en que necesitemos aplicar parches al paquete.

Binarios

Si hemos escogido este método, debemos saber que las utilidades para manejar los paquetes varían según la distribución que tengamos.

Si tenemos RedHat

En RedHat podemos instalar nuestros paquetes de dos formas distintas la primera pasa por bajar el paquete y sus dependencias (un paquete puede necesitar que otros paquetes estén instalados antes que él) de internet e instalarlas con el comando **rpm**, y la otra forma es instalar una utilidad llamada **apt-rpm** que bajará automáticamente el programa y las dependencias que se necesiten y las instalará automáticamente.

*Instalando RPMs con **rpm***

No se pretende que este manual sutituya ni mucho menos a cualquier documentación escrita de **rpm**. **rpm** es extenso y no se puede tratar en una subsección de este manual; lo mejor es que le eches un vistazo a la página del manual si necesitas algo más avanzado. Veamos unos ejemplos:

```
# rpm -ivh mutt-1.4.1-1.i386.rpm ❶
# rpm -Uvh mutt-1.5.1-1.i386.rpm ❷
```

❶ Instala el paquete mutt-1.4.1. Para que funcione tienes que tener el RPM en el mismo directorio en el que se ejecuta el comando.

❷ En caso de que mutt ya estuviera instalado en el sistema, lo mejor es correr este comando puesto que se mantendrán algunos ficheros de configuración. Se trata del comando para actualizar software.

Otras opciones útiles a la hora de instalar con **rpm** son **--force** y **--nodeps**. La primera es útil para reinstalar un paquete cuando el sistema nos dice que está instalado pero falta algún fichero; y la segunda sirve para que **rpm** instale el paquete ignorando sus dependencias. Esto último resultará en que el software no funcionará en la mayoría de los casos, pero en el futuro podrás descubrir su utilidad por tí mismo. Estas dos opciones se pueden añadir en conjunto con las ya vistas para instalar o actualizar un paquete.

Tanto para RedHat como para distribuciones basadas en esta, es posible encontrar los RPMs en los discos de instalación bajo el directorio `RedHat/RPMs/`, sustituyendo RedHat por el nombre de la distribución si no se trata de RedHat Linux.

Un sitio web muy común es rpmfind.net (http://www.rpmfind.net). Allí se puede encontrar un buscador de paquetes RPM que puede sacarnos alguna vez de un apuro si necesitamos resolver alguna

dependencia rara. No obstante, debemos tener cuidado con lo que instalamos, es recomendable leer las consideraciones de seguridad al final de este capítulo.

Puedes averiguar si un paquete está instalando con el comando **rpm -q paquete**, donde **paquete** es el nombre del paquete sin versión, por ejemplo **rpm -q sendmail** nos diría si sendmail está instalado en nuestro sistema.

*Instalando RPMs con **apt-rpm***

Lo primero será bajar e instalar el software necesario de http://apt.freshrpms.net (http://apt.freshrpms.net) , para despues instalarlo.

Una vez tengas instalado apt debes modificar el fichero `/etc/apt/sources.list` para que quede más o menos así (dependiendo la versión de tu distribución):

```
# Red Hat Linux 9
rpm http://ayo.freshrpms.net redhat/9/i386 os updates freshrpms
rpm-src http://ayo.freshrpms.net redhat/9/i386 os updates freshrpms

# Red Hat Linux 8.0
#rpm http://ayo.freshrpms.net redhat/8.0/i386 os updates freshrpms
#rpm-src http://ayo.freshrpms.net redhat/8.0/i386 os updates freshrpms
```

Una vez esté todo bien tendremos a nuestra disposición (entre otros) estos comandos:

```
# apt-get update ❶
# apt-get install paquete ❷
# apt-get upgrade ❸
# apt-cache search palabraclave ❹
```

❶ Esto actualiza la lista local de paquetes disponibles. Cada vez que vayas a instalar un nuevo paquete es interesante que ejecutes este comando.

❷ Esto instala un nuevo paquete. APT bajará todas las dependencias y las instalará. Simplemente el nombre del paquete, sin versiones ni extensiones.

❸ Este comando actualiza todos los paquetes que haya nuevos en internet. Previamente tendremos que haber actualizado nuestra lista local con 'apt-get update'.

❹ Este comando ofrecerá una lista de paquetes con una pequeña descripción que coincidan con el criterio de búsqueda *palabraclave* .

Si queremos actualizar un solo paquete y sus dependencias, primero actualizaríamos la lista local de paquetes, y después haríamos **apt-get install paquete** (sin número de versión). Si existiese una versión más reciente de este paquete se instalaría.

Tanto **rpm** como **apt-rpm** modifican la misma base de datos de paquetes RPM cuando instalan o desinstalan paquetes. La página *man* de **rpm** puede darte más información acerca de las opciones disponibles.

Hay una guía más detallada de esto en: http://linuxparatodos.com/linux/intro-apt-rpm/ (http://linuxparatodos.com/linux/intro-apt-rpm/)

Si tenemos Debian

Si nuestra distribución es Debian o alguna basada en esta, lo mejor que podemos hacer es utilizar **apt-get** para instalar software. Esta distribución utiliza también paquetes binarios, pero son distintos de los RPMs y se llaman comúnmente DEB.

*Instalando DEBs con **dpkg***

Aunque esta manera de instalar software en Debian es un poco desaconsejada mostraré el comando para instalar un paquete **.deb** que hayamos descargado de internet. **dpkg** tiene un cometido similar al de **rpm** en las distribuciones basadas en RedHat a diferencia de que **dpkg** como es lógico sólo trabaja con paquetes binarios DEB.

```
# dpkg -i paquete.deb
```

*Instalando DEBs con **apt-get***

El caso de **apt-get** es idéntico al de **apt-rpm**. Los comandos que podremos usar son (entre otros):

```
# apt-get update ❶
# apt-get install paquete ❷
# apt-get upgrade ❸
# apt-cache search palabraclave ❹
```

❶ Esto actualiza la lista de paquetes local. Cada vez que vayas a instalar un nuevo paquete es interesante que ejecutes este programa.

❷ Esto instala un nuevo paquete. APT bajará todas las dependencias y las instalará. Simplemente el nombre del paquete, sin versiones o extensiones.

❸ Este comando actualiza todos los paquetes que haya nuevos en internet. Previamente tendremos que haber actualizado nuestra lista local con 'apt-get update'.

❹ Este comando ofrecerá una lista de paquetes con una pequeña descripción que coincidan con el criterio de búsqueda *palabraclave* .

Exactamente igual que ocurría con **apt-rpm**, **apt-get** nos permite actualizar un paquete previa actualización de nuestra lista local con **apt-get install paquete**. También, el fichero `/etc/apt/sources.list` contiene las direcciones e información de dónde bajar los paquetes. Es muy común en las distribuciones Debian (también lo es con apt-rpm pero no tanto), añadir nosotros mismos líneas al `sources.list` para poder instalar mediante **apt** paquetes que no están en las fuentes oficiales (nuevas versiones generalmente). Después de añadir (o quitar) líneas al `sources.list` se debe actualizar la lista de paquetes local.

La base de datos de paquetes DEB

Tanto la instalación/desinstalación de paquetes con **dpkg** como con **apt**, registran sus cambios en la misma base de datos de paquetes Debian. **dpkg** es el programa que permite hacer distintas operaciones con esta base de datos. Su página *man* te dará más detalles.

Si tenemos Slackware

Los paquetes en Slackware usan un sistema bastante simple. Un paquete de Slackware es un paquete tar comprimido con gzip (Ver Capítulo 12). Descomprimiendo a mano uno de estos paquetes en el directorio raíz (/) los contenidos de aquél se distribuirían adecuadamente en el sistema (por ejemplo los binarios en `/usr/bin`, los ficheros de configuración en `/etc` ...). Pero además del software en sí estos paquetes contienen información extra, útil para las utilidades que gestionan los paquetes binarios de esta distribución.

Esto no debe llevar a confusión con los paquetes de código fuente; un paquete binario de Slackware es justamente eso, aunque venga comprimido como habitualmente descargamos los paquetes de código fuente.

La nomenclatura de los paquetes es del tipo `nombre-version-plataforma-IDcompilación.tgz` por ejemplo `cdrdao-1.1.7-i386-1.tgz`

> **Aviso**
>
> Las utilidades básicas de paquetes de Slackware NO GESTIONAN DEPENDENCIAS ENTRE PAQUETES, esto es, no se quejarán si se instala un paquete que depende de otro que no está instalado. Obviamente el software del primer paquete no funcionará.
>
> Pero es posible gestionarlas con otras utilidades descritas en la sección la sección de nombre *Gestión de dependencias*. Además, como irás descubriendo si usas habitualmente Slackware, esto no es una desventaja en todos los aspectos como pudiera parecer al principio.

Instalando paquetes normalmente

La utilidad **installpkg** instala un paquete en el sistema. Un ejemplo sería, suponiendo que hemos descargado el fichero de Internet o estamos en el directorio del CDROM donde aquél se encuentra:

```
# installpkg cdrdao-1.1.7-i386-1.tgz
```

Opciones interesantes para esta órden son:

- **-warn** no instala el paquete pero nos informa de lo que pasaría si lo instalamos (ficheros nuevos o sobreescritos).
- **-r** instalará los paquetes de el directorio de trabajo y además recursivamente los que haya en directorios inferiores que coincidan con el nombre de archivo que, además, puede contener los *wildcards* ya conocidos.

Más opciones de esta órden en su página **man**

Actualizando paquetes

Si tenemos un paquete instalado y nos hemos hecho con una versión actualizada, debemos usar la órden **upgradepkg <nombrefichero>** para actualizar al nuevo nuestro sistema.

Eliminando paquetes

La órden **removepkg** seguida del nombre del paquete a quitar del sistema (puede ser sólo el nombre, sin versión, pero también el nombre del fichero completo que se usó para instalarlo -útil si hay varias versiones-).

Del mismo modo que **installpkg**, aquí también disponemos de la opción **-warn** que nos permite ver qué cambiaría en nuestro sistema sin desinstalar el paquete realmente.

***pkgtool**, utilidad con menús*

Ejecutando esta orden sin argumentos en un shell como root, obtendremos un menú con diversas opciones que hablan por sí solas. Nos permitirá intuitivamente instalar, desinstalar, actualizar, ver información o los paquetes instalados, etc.

Conversor de paquetes RPM a TGZ

Esta posibilidad nos será útil si tenemos un paquete RPM que no tiene muchas dependencias (porque si las tiene, las tendremos que satisfacer copiando librerías o similar manualmente, y las rutas pueden ser distintas entre las distribuciones) y no podemos conseguir un paquete Slackware y además no nos apetece compilarlo.

Si las condiciones anteriores se cumplen, la orden **rpm2tgz fichero.rpm** nos generará un paquete .tgz que podremos instalar normalmente con las utilidades concidas.

Gestión de dependencias

Internamente Slackware sí que conoce que paquetes dependen de cuáles otros, así que hay utilidades que nos permiten instalar y desinstalar gestionando las dependencias entre paquetes para que nosotros nos olvidemos de ello.

Si no la mejor, una de las mejores es swaret (http://swaret.sourceforge.net). Para hacerla funcionar, simplemente instala el paquete que puedes descargarte desde la dirección anterior y edita el fichero `/etc/swaret.conf`

Lo único que hay que ajustar en este fichero es el número de versión de Slackware que se tiene, y luego descomentar (borrar los #) de las líneas que comienzan por ROOT que nos interesen. Estas líneas representan fuentes desde las cuales se descargarán los paquetes que se quieran instalar (y sus dependencias).

A continuación de lo anterior se pueden añadir repositorios de paquetes no oficiales (como Linux-Packages (http://www.linuxpackages.net)) mediante las líneas REPOS_ROOT. Para más opciones consultar los comentarios del resto del fichero.

Para instalar un paquete:

```
# swaret --update
# swaret --install cdrecord
```

La primera línea actualiza el listado de paquetes con los más recientes y la segunda descarga e instala el paquete `cdrecord`. Si hay varias versiones de este paquete disponibles se nos preguntará una por una si deseamos o no instalarla.

Hay muchas más cosas que esta herramienta puede hacer, hay más información en su página **man**.

SlackBuilds y creación casera de paquetes

Uno de los principales problemas de instalar paquetes de código fuente es que en algunos casos puede no ser inmediata su desinstalación. Una posible solución es compilar el paquete de código y a partir de estos binarios crear un paquete Slackware "casero".

Este proceso se escapa al nivel del resto de este capítulo y viene detallado perfectamente en LinuxPackages.net (http://www.linuxpackages.net) en los documentos *Building Packages* y *The Perfect Package*; y hay algunas notas en Slackware Linux Essentials (http://www.slackware.com/book), libro oficial. Al final la utilidad **makepkg** (ver su página man) se utiliza para crear el paquete.

Es muy habitual que esos pasos se sintenticen en un script de shell (distinto para cada paquete). Son lo que se conocen como *SlackBuilds* y es de este modo como se crean los paquetes oficiales de Slackware Linux. Se pueden encontrar algunos de ejemplo en LinuxPackages y también en el directorio de fuentes ("source") de las releases de Slackware Linux.

Antes de disponerte a crearte el tuyo busca bien no sea que alguien haya hecho ese trabajo por tí (es bastante probable).

Si tenemos Mandrake

Esta distribución trae consigo una utilidad propia, **urpmi**, que sirve para manejar las instalaciones de software. Mandrake guarda gran similitud con Red Hat, y de hecho está basada en ella. Así, **urpmi** es el equivalente al **apt-get** de Red Hat, y como en el caso de Red Hat, **urpmi** instala automáticamente el paquete elegido y resuelve sus dependencias. Además, sigue siendo posible instalar el paquete seleccionado y sus dependencias a mano con la utilidad **rpm** ya que Mandrake la trae de serie.

*Instalando RPMs con **rpm***

Aquí nos encontramos en una situación exáctamente igual a la de Red Hat. Para instalar y actualizar paquetes, podemos usar:

```
# rpm -ivh mutt-1.4.1-1.i386.rpm ❶
# rpm -Uvh mutt-1.5.1-1.i386.rpm ❷
```

❶ Instala el paquete mutt-1.4.1. Para que funcione tienes que tener el RPM en el mismo directorio en el que se ejecuta el comando.

❷ En caso de que mutt ya estuviera instalado en el sistema, lo mejor es correr este comando puesto que se mantendrán algunos ficheros de configuración. Se trata del comando para actualizar software.

El **rpm** de Mandrake (así como el de otras distros) aceptará exactamente las mismas opciones que están disponibles en el **rpm** de RedHat Linux.

*Instalando RPMs con **urpmi***

Podremos apreciar la similitud de funcionamiento entre las distintas utilidades. El caso de **urpmi** es casi idéntico al de **apt-get**, pero aquí las fuentes predeterminadas son los CDs de instalación. Los comandos que podremos usar son, entre otros:

```
# urpmi.update -a ❶
```

```
# urpmi paquete ❷
# urpmi --auto-select ❸
```

❶ Esto busca actualizaciones dentro de las fuentes de software que se encuentren en unidades no removibles. Si además de los CDs de fuentes que trae por defecto mandrake, tiene alguna fuente que pueda haber sido actualizada, es interesante ejecutar este comando antes de instalar un paquete.

❷ Este instala un nuevo paquete. **urpmi** buscará entre las fuentes donde conseguir el paquete y sus dependencias. Si necesita conseguir algún paquete de una unidad removible, como puede ser el CDROM, nos pedirá que insertemos el disco correcto.

❸ Este comando actualiza todos los paquetes que haya nuevos listados en las fuentes.

Añadiendo una nueva fuente de software

A medida que pasa el tiempo, van saliendo nuevas versiones de paquetes constantemente, y seguramente nos interesará actualizar algunos paquetes concretos, por ejemplo, aquellos que más utilicemos.

Para ello Mandrake tiene una rama llamada en desarrollo que es la que cuenta con las versiones más nuevas de los paquetes disponibles, aunnque sean más propensos a contener errores al tratarse de una versión de Mandrake en desarrollo; Mandrake Cooker. Mandrake dispone de diferentes mirrors o servidores réplica (espejos) que contienen una lista actualizada de los paquetes de Mandrake Cooker, que nosotros usaremos como fuente de software.

La sintaxis del comando para añadir una nueva fuente de software es la siguiente:

```
# urpmi.addmedia [opciones] <nombre> <dirección> [with hdlist.cz]
```

No vamos a entrar en que opciones se le pueden dar como argumento al comando **urpmi.addmedia** pues se sale fuera de contexto. El nombre es como llamaremos a esta nueva fuente de software, y la dirección es el lugar donde se encuentra localizada la misma. Opcionalmente, podemos indicar explícitamente cual es el fichero que contiene la lista de los paquetes que se hallan dentro del repositorio de software.

El fichero `hdlist.cz` es por defecto el que conitene la lista de paquetes del repositorio, pero normalmente existe una lista de un tamaño increiblemente menor que nos sirve para el mismo fin pero que nos ahorrará tiempo de descarga si la fuente de software se encuentra en Internet, y es un archivo normalmente llamado `synthesis.hdlist.cz`.

Con todo, el comando para añadir una nueva fuente de software mandrake cooker podría ser el siguiente, cambiando si quieres el servidor espejo por el que prefieras:

```
# urpmi.addmedia ftp-cooker
ftp://ftp.ciril.fr/pub/linux/mandrake-devel/cooker/cooker/Mandrake/RPMS with synthesis.h
```

Creando nuestro propio repositorio de software

Posiblemente necesitemos descargar e instalarnos paquetes RPM que ni siquiera se encuentren dentro del repositorio de software de Mandrake Cooker. Para eso hay una solución: añadir una nueva fuente

de software localizada en un directorio de nuestro disco duro, que es donde meteremos todos aquellos paquetes que nos descarguemos.

De esta manera luego podremos manejar estos paquetes mediante **urpmi**.

Pues bien, los pasos para crear nuestro repositorio de software son los siguientes:

```
# mkdir -p /usr/local/repositorio ❶
# cd /usr/local/repositorio ❷
# wget -c http://belnet.dl.sourceforge.net/sourceforge/gaim/gaim-0.66-1mdk9.1.i586.rpm ❸
# genhdlist . ❹
# urpmi.addmedia repositorio_local file://usr/local/repositorio with synthesis.hdlist.cz
# urpmi gaim ❻
```

❶❷ Cremos el directorio donde se situará el repositorio de software y nos situamos dentro de él. Naturalmente si ya tenemos un directorio para el repositorio no necesitamos crearlo, pero si entrar dentro de él.

❸ Para seguir necesitamos tener algún paquete RPM dentro del repositorio de software, así que optamos por bajarnos el paquete gaim (para la mensajería instantánea), pero puedes o meter paquetes que ya tienes bajados en otro lugar, o optar por descargar otro(s) paquetes(s) de software en formato RPM, eso sí.

❹ Este paso es muy importante, ya que aquí creamos la lista de paquetes que urpmi necesita para poder usar este repositorio como fuente de software.

Has de saber que si cambias algo dentro del repositorio (añades, borras o actualizas un paquete) si quieres que urpmi tenga su base de datos de paquetes actualizada has de volver a ejecutar este comando dentro del directorio del repositorio.

❺ Con este comando dejamos listo y en funcionamiento nuestra flamante y completa fuente de software a la lista de fuentes de Mandrake.

❻ Finalmente podemos comprobar que los paquetes que tenemos en el repositorio son visibles e instalables mediante urpmi, ¡enhorabuena!

Si tenemos Gentoo

Instalando ebuilds del portage

Si te las has arreglado para instalar esta distribución, es más que probable que ya sepas como instalar y desinstalar software, pero vamos a describirlo superficialmente.

Podríamos definir *portage* como el árbol o lista de paquetes disponibles para Gentoo Linux; lo que más o menos equivaldría a la *lista de paquetes local* que nombrábamos para otras distribuciones. Si quieres instalar el último software, tendrás que actualizar esta lista:

```
# emerge sync
```

emerge es la herramienta que utilizaremos principalmente para instalar software en Gentoo. Puedes leer su página **man** para más detalles de los explicados aquí. Tras introducir este comando se descargará la nueva lista de ebuilds.

Para actualizar todos los paquetes que tengan nuevas versiones, tenemos:

```
# emerge -up system
# emerge -u system
```

La primera línea mostrará los paquetes que se actualizarían, y la segunda los bajaría y los instalaría.

Si queremos instalar un paquete que está en el árbol de ebuilds (portage). Instalaremos el programa con:

```
# emerge paquete
```

Exactamente igual que ocurría con apt, emerge bajará, compilará (en su caso) e instalará las dependencias del paquete y luego el paquete. Si queremos ver cuáles son las dependencias de un determinado paquete, podemos usar la opción **-p**:

```
# emerge -p paquete
```

Podemos buscar paquetes por palabras clave con la opción **-s** :

```
# emerge -s palabraclave
```

Un *ebuild* es uno de los meta-paquetes que instalamos con emerge. Generalmente el software será compilado en nuestra máquina, lo que implica una gran diferencia con los sistemas de paquetes descritos anteriormente; y es que un paquete puede ser compilado con soporte o sin soporte para otro software distinto. Compilarlo con soporte para este otro software, dará al paquete funcionalidades extra. La mayoría de las veces nos interesará activar el soporte de un paquete para el mayor número de programas que tengamos instalados. Aquí entran en juego dos elementos muy importantes del sistema de paquetes de Gentoo: el fichero `/etc/make.conf` y la variable `USE`, definida en este mismo fichero. Esta variable contendrá los paquetes para los que queremos que se habilite soporte cuando compilemos otros. Hay multitud de documentos que describen todo esto en la web de documentación de Gentoo Linux (http://www.gentoo.org/doc).

Por último, podemos actualizar un solo paquete (y sus dependencias mediante **emerge -u paquete**, donde *paquete* es simplemente el nombre del paquete sin versiones ni extensiones.

Instalando ebuilds no oficiales

Si lo que queremos es instalar un paquete que no está en el arbol de Gentoo (portage), lo primero que tendremos que hacer es ajustar la variable `PORTDIR_OVERLAY` a `/usr/local/portage`. Para esto tendremos que editar el fichero `/etc/make.conf`.

Una vez hemos ajustado la variable `PORTDIR_OVERLAY`, creamos el directorio, la categoría y corremos unos comandos que son necesarios para evitar instalar programas troyanizados. Se supone que queremos instalar ssmtp que está en la categoría net-mail.

```
# mkdir -p /usr/local/portage ❶
# cd /usr/local/portage ❷
# mkdir -p net-mail/ssmtp/files ❸
# cd net-mail/ssmtp ❹
# cp /donde/este/ssmtp-2.60.3.ebuild ./ ❺
# ebuild ssmtp-2.60.3.ebuild digest ❻
# ACCEPT_KEYWORDS="~x86" emerge ssmtp ❼
```

❶❷ Creamos el directorio donde pondremos los ebuilds y cambiamos allí

❸❹❺ Creamos la categoría y el directorio del paquete y cambiamos a este último. Además copiamos allí el ebuild.

❻ Este paso es muy importante, se crean unos ficheros con hashes md5 para evitar que instalemos programas troyanizados y cosas así.

❼ Con este comando finalmente portage instala el paquete en cuestión.

Caso especial, alien

Hay ocasiones en que un paquete de software no está disponible para el sistema de paquetería de nuestra distribución. En estos casos la alternativa a usar tiene nombre propio y se llama *alien*. Alien es un programa que transforma los paquetes de un sistema a otro.

Como ejemplo generaremos un RPM a partir de un DEB

```
# alien --to-rpm mutt-1.5.4.deb
```

Opcionalmente se le puede pasar el flag **-i** para que alien instale el paquete generado automáticamente.

Alien se encuentra en: http://kitenet.net/programs/alien/ (http://kitenet.net/programs/alien/)

Fuentes

En caso de que vayamos a compilar un programa por nuestra cuenta lo mejor es que leamos los ficheros de ayuda `README` e `INSTALL` del paquete. Para descomprimirlo haremos (suponiendo el archivo de fuentes descargado en el directorio actual):

```
# tar zxf paquete-version.tar.gz
```

Si el paquete es un .tar.bz2 haremos:

```
# tar jxf paquete-version.tar.bz2
```

Los pasos "genéricos" son:

```
# ./configure
# make
# make install
```

Pero como las cosas pueden cambiar entre unos paquetes y otros lo mejor que podemos hacer es leernos los ficheros de documentación que acompañen a dicho programa.

> **Nota:** Para más información acerca de qué es un fichero `.tar.gz`, `.tar.bz2` y cómo trabajar con ellos, se explican detalladamente en Capítulo 12.

Desinstalando lo instalado

Binarios

En las diferentes sub-secciones pondremos el comando genérico que hay que ejecutar si queremos desinstalar un paquete que ya esté instalado. A veces se nos avisará que otros paquetes dependen de el paquete que queremos quitar; en ese caso, lo más sencillo es quitar estos paquetes primero y después desinstalar el que queríamos quitar.

RPMs

```
# rpm -evh paquete
```

En caso de usar apt-get para RPMs tambien disponemos del comando:

```
# apt-get remove paquete
```

DEBs

```
# apt-get remove paquete
```

TGZs

```
# removepkg paquete
```

EBUILDs

```
# emerge -C paquete
```

Fuentes

En el directorio donde se descomprimieron las fuentes ejecutamos:

```
# make uninstall
```

Utilidades Gráficas

En el caso de las distribuciones que usan paquetes binarios, podemos muchas veces encontrar una utilidad gráfica para instalar y desinstalar paquetes en los menús de nuestros escritorios. Para poder usar estas aplicaciones (que varían en nombre y funcionamiento para cada distribución o versión de la misma) se nos pedirá la contraseña de **root**. Este tipo de aplicaciones puede sernos útil en nuestras primeras etapas con Linux si necesitamos instalar un paquete y no tenemos tiempo de mirar las opciones. Si quieres aprender a usar Linux, este tipo de aplicaciones simplemente serán un recurso de uso espontáneo. Lo mejor es familiarizarse con las utilidades comentadas.

Consideraciones sobre seguridad

Venimos aprendiendo que Linux es un sistema seguro dado que es multiusuario, y por eso los usuarios no privilegiados (usuarios normales) no pueden dañar la máquina. Aunque discutiremos el tema de la seguridad ampliamente en secciones posteriores, dado lo anterior no es posible que un usuario normal pueda instalar un virus (programa dañino) en nuestra máquina.

Hasta aquí todo bien, pero hay que darse cuenta de que para instalar software en la máquina debemos ser **root**. Esto quiere decir que instalemos el paquete de software que instalemos, no se nos impondrá ninguna restricción. El único problema está en que algún paquete de software que queramos instalar puede contener virus o programas espía.

¿Cuál es la solución a este problema? La única solución a este problema es instalar SIEMPRE SOFTWARE DE SITIOS OFICIALES O CONFIABLES. De esta forma sabemos que el software que instalamos es seguro.

Quizás uno se pueda preguntar... ¿no solucionan los antivirus este problema? La respuesta es NO. Existen antivirus para Linux pero su funcionalidad no suele ser esta. Se discutirá más adelante sobre los antivirus.

Sistemas de paquetes y manejo de librerías

Las librerías son partes de código usados por muchos programas diferentes. Por esto, muchas veces algunos paquetes requerirán otros paquetes del tipo `lib*`. Todos los ficheros de librerías que se instalan en el sistema comienzan su nombre por *lib* debido a un antiguo convenio UNIX.

Los sistemas de paquetes binarios de las distribuciones (RPM y DEB) hacen una distinción con las librerías. Podríamos decir que un mismo conjunto de librerías, lo dividen en dos paquetes, por ejemplo, `libncurses` y `libncurses-devel` (o `libncurses-dev` para Debian). Para instalar software mediante paquetes que necesiten esta librería, bastaría con instalar generalmente sólo el primer paquete. El segundo paquete (*-dev* en DEBs o *-devel* en RPMs), se conoce como *ficheros de desarrollo* (development), y es necesario instalarlo si, por ejemplo, vamos a compilar desde el código fuente un paquete que necesite *libncurses*. Así, es perfectamente posible que haciendo un **./configure** para instalar desde las fuentes se nos avise de que nos falta *libncurses* si solamente tenemos el primer paquete instalado.

ldconfig y más sobre librerías

Las librerías compiladas (ya en código de máquina), están en `/usr/lib` si se instalaron desde paquetes de la distribución, y en `/usr/local/lib` si se instalaron desde fuentes, o un subdirectorio

de estos dos, (aunque con **./configure --help** descurbriremos opciones que nos permiten especificar también dónde queremos que se instalen tanto los programas como las librerías). Su nombre suele ser `libNOMBRE.so`, o bien `libNOMBRE.so.VERSION`. La extensión *.so* es simplemente un convenio y es abreviatura de Shared Object (Objeto Compartido, ya sabemos por qué).

En los sistemas Linux, la utilidad encargada de manejar la instalación de las librerías y proveer al resto de programas información sobre su localización en el sistema de ficheros es **ldconfig**. Esta utilidad tiene un fichero de configuración, `/etc/ld.so.conf`, donde se listan los directorios que contienen librerías, por ejemplo:

```
# cat /etc/ld.so.conf
/usr/kerberos/lib
/usr/X11R6/lib
/usr/lib/qt-3.1/lib
/usr/lib/mysql
/usr/lib/kde3
/usr/local/lib
/usr/lib/wine
/usr/lib
```

Aunque muchas veces la instalación lo hará por nosotros, cuando instalemos nuevas librerías (especialmente desde código fuente), es una buena costumbre añadir el directorio donde se han copiado las librerías a este fichero de configuración (si no está ahí), y correr como root el comando **ldconfig**. Si no lo hacemos así, es posible que al intentar instalar otros programas desde fuentes, se nos diga que no puede encontrar tal o cuál librería.

Capítulo 9. Otros comandos útiles

Introducción

Ya hemos visto algunos comandos útiles en los capítulos anteriores. Como nunca dejaremos de aprender comandos nuevos, aquí van otros cuantos más que conviene conocer. La descripción que se hace aquí de ellos es bastante básica, pero siempre puedes recurrir a su página **man**. Además, en este capítulo se explican cosas interesantes como la gestión de memoria en Linux.

Comandos relacionados con la E/S

De estos comandos hemos conocido ya unos cuantos en secciones anteriores: **echo**, **cat**, **more**, **less** y las redirecciones más comunes. Ahora veremos algunos más:

head y tail

head sirve para mostrar (o redirigir a otra parte) el principio de un fichero o de la información proveniente de un *pipe* o tubería. **tail** tiene exactamente el mismo cometido, con la diferencia de que este último, lo que muestra es el final. Ejemplos:

```
$ head mifichero ❶
$ head -n 30 mifichero | less ❷
$ tail mifichero ❸
$ tail -n 35 mifichero > final_de_mifichero ❹
```

❶ Mostrará por pantalla las primeras 10 líneas de `mifichero`

❷ La opción `-n` se usa para indicar a continuación el número de líneas que se deben mostrar desde el principio del fichero (o como en este caso, enviar a **less**).

❸ Cumple exactamente la misma función que **head**, a diferencia de que mostrará las 10 últimas líneas del fichero.

❹ Escribirá las 35 últimas líneas de `mifichero` en el fichero `final_de_mi_fichero`.

Además de leer desde un fichero, estos dos comandos pueden leer desde `stdin`, así que podemos "conectarlos" a pipes o tuberías. Supongamos que estamos instalando un programa desde un paquete de código fuente, y que ejecutamos **make**. Imaginemos que por cualquier motivo, **make** produjese una larguísima salida de errores, tantos que no pudiésemos ver cuál es el primero de ellos. Pero nosotros necesitamos saber cuál fue el primer error emitido para poder solucionar el problema. Así que podríamos hacer:

```
$ make 2>/dev/stdout 1>/dev/null | head -n 20
```

Atención a la primera parte del comando. Recordando que las redirecciones de este tipo se procesan de derecha a izquierda (las de 1>... 2>...), lo que hacemos primero es despreciar la salida normal, y posteriormente, enviar `stderr` (la salida con errores) a la salida estándar, que con el pipe se convierte en la entrada estándar del comando **head**, que lee automáticamente de ella, obteniendo así nuestro

resultado. (Nota que si hubiéramos puesto las redirecciones cambiadas de sitio, no habríamos obtenido salida alguna.)

Una opción realmente útil de **tail** es la opción `-f`. Con ella, **tail** escribirá el final del fichero en `stdout` y además permanecerá activo. Si algo se añade al fichero desde otro sitio también lo mostrará. Por ejemplo, abre una terminal y escribe:

```
$ touch fichero
$ for i in a b c d e f; do echo $i >> fichero; done
$ tail -f fichero
a
b
c
d
e
f
```

Mantén esta terminal abierta y pásate a otra terminal. En esta otra escribe:

```
$ echo "hooooooola" >> fichero
```

Vuelve ahora a la primera terminal y observa como `hooooooola` también ha sido sacado por pantalla por **tail -f**. Esta opción es especialmente útil para examinar ficheros de log (registro de actividad) del sistema mientras hacemos cambios en él como veremos en su momento.

El comando cut

cut tiene como uso principal mostrar una columna de una salida determinada. La opción `-d` va seguida del delimitador de los campos y la opción `-f` va seguida del número de campo a mostrar. Por ejemplo:

```
$ cat /etc/passwd | head
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:100:sync:/bin:/bin/sync
games:x:5:100:games:/usr/games:/bin/sh
man:x:6:100:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
$ cat /etc/passwd | head | cut -d ":" -f 1,7
root:/bin/bash
daemon:/bin/sh
bin:/bin/sh
sys:/bin/sh
sync:/bin/sync
games:/bin/sh
man:/bin/sh
lp:/bin/sh
mail:/bin/sh
news:/bin/sh
$ cat /etc/passwd | head | cut -d ":" -f 1
root
```

```
daemon
bin
sys
sync
games
man
lp
mail
news
```

El "delimitador" por defecto es el tabulador, nosotros lo cambiamos con la opción `-d`. Tiene algunas otras opciones útiles, consulta su página **man**.

Algunos otros comandos relacionados con la E/S

El uso que harás de estos comandos será más bien esporádico, por lo que simplemente los citamos. En sus respectivas páginas *man* encontrarás opciones específicas para ellos.

- **wc** (Word Count, contador de palabras). Muestra el número de palabras, bytes y caracteres que contiene un fichero determinado.
- **strings**. Muestra por pantalla los caracteres "imprimibles" de los ficheros binarios.
- **tac**. Hace exactamente lo mismo que **cat** pero muestra las líneas en orden contrario (desde la última hasta la primera).
- **sort**. Muestra el contenido de un fichero, pero mostrando sus líneas en orden alfabético.

Comandos relacionados con la memoria y el disco

Es posible que no se entienda alguna cosa; más adelante en el capítulo de *dispositivos* se despejarán todas las dudas.

df

df (Disk Free) nos informa del espacio disponible de las unidades de disco (locales o de red) que tengamos montadas.

```
# df
S.ficheros           1K-blocks      Used Available Use% Montado en
/dev/hdd1              6040288   3247924   2485528  57% /
/dev/hda2             11719052   3526412   8192640  31% /home
none                    128008         0    128008   0% /dev/shm
```

Una opción útil de la que dispone es **-m**, que mostrará el espacio en MegaBytes.

Gestión de memoria RAM en Linux y free

En primer lugar, entendamos cómo se gestiona la memoria física del sistema en Linux. El kernel tiende a tomar, primeramente, la memoria que necesitan los procesos que corre. Conforme el sistema está en marcha más tiempo, el kernel toma prácticamente la totalidad de la memoria física existente, dejando sólamente unos cuantos MB de memoria físicamente libres. Muchas veces esto lleva a los principiantes en Linux a confusión, llegando a creer que la gestión de la memoria que hace Linux no es eficiente.

¿Y para qué usa esa memoria restante Linux? Pues la usa como *buffers*, esto es, guarda datos para un más rápido acceso al disco duro, datos de programas que se abrieron, por si se vuelven a abrir, que se invierta mucho menos tiempo en ello, etc. En definitiva, aprovecha la memoria físicamente libre para agilizar tareas básicas. Por otro lado esto es bastánte lógico... ¿para qué te sirve tener una gran cantidad de memoria RAM física libre? ¿Y si la usamos para hacer que el sistema vaya más rápido? Mucho mejor ;-), ¿no te parece?

Si la carga de tareas (procesos) que el sistema tiene que soportar lo carga y estos procesos (o programas ejecutándose) necesitan esta memoria que se está usando como buffers, Linux borra estos buffers de la memoria y la asigna a las nuevas tareas que la necesiten.

En definitiva, ¿cómo podemos saber la memoria que tenemos libre y lo que ocupa cada cosa (procesos por un lado y buffers por otro)? La respuesta la tenemos en el comando **free** (la salida por defecto es en KBs).

```
# free

      total       used       free     shared    buffers     cached
Mem:        256016     251304       4712          0      12236     177516
-/+ buffers/cache:      61552     194464
Swap:       409648       2964     406684
```

En el equipo de este ejemplo, hay 256 MB de RAM física instalada. De ellos, solamente 4 MB están libres (de lo que deducimos que la máquina lleva tiempo encendida y que se han escrito buffers en parte de la RAM). En la línea `-/+ buffers/cache` tenemos en *used* la cantidad de memoria estimada que los procesos del sistema están usando, y en *free*, la suma de la memoria usada para buffers y caché más la físicamente libre. Así, los procesos sólo están necesitando 60 MB de RAM, y el resto de memoria usada está simplemente agilizando el sistema.

Ahora nos fijamos en la última línea. Ahí tenemos el uso de la partición SWAP de intercambio de datos. Esta es una buena medida para saber lo "cargado" que está nuestro sistema. En este caso hay escritos menos de 3 MB en la memoria SWAP, lo que pone de manifiesto que la máquina anda holgada. Más tarde, en el capítulo de procesos descubriremos otros datos que nos permitirán hacernos una idea de la carga del sistema.

du, uso del espacio de disco

El comando **du** nos indica el espacio que ocupa un directorio del sistema de ficheros y todo lo que tiene debajo de él.

Es tan fácil de usar como útil: **du directorio/**.

Con él podemos comprobar dónde se acumula la mayor cantidad de espacio en disco de nuestro sistema.

mc

Este es un programa que ofrece una interfaz de usuario basada en texto bastante cómoda, generalmente para hacer operaciones con ficheros (especialmente réplicas de un directorio y todo lo que haya debajo de él o copias grandes de una gran parte del sistema de ficheros), o tener una "vista de pájaro" del sistema de ficheros.

Su uso es intiutivo (y todavía más sencillo si tenemos habilitados los servicios de ratón en la consola), y por ello no merece mucha más explicación. Recuerda este comando porque nos será útil en el futuro.

file

Este comando nos indicará qué tipo de datos contiene el fichero que le pasemos como primer argumento.

Comandos útiles varios

Algunos otros comandos útiles varios que no encajan en las categorías de arriba. Los que no queden descritos en esta sección, es que vendrán explicados con detalle en alguna sección posterior.

gcc, el compilador de C

Probablemente profundicemos un poco más en él próximamente, pero por si quieres hacer alguna cosa en C y no sabes todavía cuál es el compilador en Linux, **gcc** es el GNU C Compiler. Si no lo tienes, necesitarás instalarlo desde los discos de tu distribución. La mayoría de los sistemas UNIX responderán al comando **cc** como C Compiler, y ejectuarán el compilador que haya instalado en el sistema, en este caso, **gcc**.

Para crear un ejecutable desde los ficheros de código C, basta con que hagamos:

```
$ gcc fichero1.c fichero2.c [...] -o miejecutable
```

uname

Indica nombre y versión del sistema operativo. Usado con la opción `-a` muestra toda la información.

```
$ uname -a
Linux hostname 2.4.21 #1 SMP Thu Sep 4 20:50:32 CEST 2003 i686 unknown
```

which

Le pasamos como primer argumento un comando, y nos dirá, de los directorios que existen en `$PATH`, en cuál de ellos está, por ejemplo:

```
$ which ls
/bin/ls
```

touch

touch cambia la fecha de creación o de modificación de un archivo. Si no existe, lo crea vacío. Lo más conveniente es leer su página del manual: **man touch**

Comandos de información sobre usuarios, tiempo y fecha

Los comandos **w** y **who** nos informarán de los usuarios que actualmente están usando la máquina, y algunos datos del tiempo que llevan usándola o los procesos que están ejecutando.

```
$ w
19:05:51 up 50 min,  4 users,  load average: 0.00, 0.02, 0.06
USER   TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
user1  :0       -                18:16   ?xdm?   0.00s   ?     -
user2  pts/0    :0.0             18:18    0.00s  0.09s  0.00s  w
user2  pts/1    :0.0             18:31    4.00s  4.08s  4.06s  vim chapter09.xml
user   pts/2    :0.0             18:33   32:03   6.90s  0.02s  -bash

$ who
ricardo  :0           Sep  5 18:16
user1  pts/0        Sep  5 18:18 (:0.0)
user2  pts/1        Sep  5 18:31 (:0.0)
user2  pts/2        Sep  5 18:33 (:0.0)
```

El comando **uptime** nos informará del tiempo que lleva la máquina encendida, además de algunos otros datos:

```
$ uptime
 19:09:19 up 53 min,  4 users,  load average: 0.00, 0.00, 0.04
```

El comando **date** mostrará la hora del sistema, y además, nos permite cambiarla con la opción `-s`.

Un comando especialmente interesante es **ntpdate** (probablemente tengas que instalarlo). Sincroniza la hora del sistema con un servidor de hora del protocolo NTP. Hay mucha información del protocolo NTP disponible en la red. Lo que nos interesa saber es que hay una serie de servidores que actualizan su reloj interno mediante dispositivos de GPS, así no sufren desviaciones de tiempo. nosotros podemos beneficiarnos de ello, con el comando **ntpdate SERVIDOR** la hora de nuestro sistema se actualizará con la de SERVIDOR. En España hay unos cuantos, como ejemplos: `hora.uv.es` `hora.rediris.es` Lógicamente, para poder ejecutar este comando y poder cambiar la hora del sistema deberemos entrar como `root`. Más tarde aprenderemos a hacer que este comando se ejecute en cada inicio o periódicamente.

El comando **whoami** muestra nuestro nombre de usuario.

Buscar archivos: find y locate

En Linux tenemos dos opciones si queremos buscar archivos en nuestros discos. Diferen en que son conceptos totalmente distintos que a su vez cumplen necesidades distintas. En primer lugar, **find** es el comando que se usa para buscar normalmente en el sistema de ficheros, y lo examina cada vez que queremos hacer una búsqueda.

El comando **locate** se complementa con el comando **updatedb**. Este último comando, cuando lo ejecutamos (sin opciones ni argumentos), crea una especie de "base de datos" de todos los ficheros y directorios del sistema. De este modo, cuando queremos buscar algo con **locate**, simplemente este programa busca en esta base de datos, lo cual es mucho más rápido que buscar en el sistema de archivos directamente. Obvimanete esto tiene un inconveniente; si no actualizamos esta base de datos con **updatedb**, la información que nos dará **locate** no será cierta.

Uso de find

El uso más común de este comando es el siguiente:

```
find <directorio> -name <nombre>
```

El primer argumento (*directorio*) es el directorio desde el que queremos que se busque (la búsqueda es recursiva por defecto, buscará también en todos los directorios que haya por debajo), la opción (*-name*) es el modo de búsqueda (en este caso por nombre), y el tercer argumento (en este caso *nombre*) son los criterios de búsqueda. Algunos ejemplos:

```
# find /usr -name mozilla
/usr/bin/mozilla
/usr/lib/mozilla
# find /usr -name '*.so'
[... casi todas las librerías del sistema ...]
# find / -name '*gtk*'
[... todos los ficheros cuyo nombre contenga "gtk"...]
# find / -size +3000k
[... todos los ficheros de tamaño >= 3000 KB ...]
```

Nota que para poder usar los wildcards que conocemos, simplemente debes encerrar el criterio entre comillas simples. Se puede buscar por criterios de muchísimos tipos, para más detalles, mira la página man. Recuerda que puedes detener la búsqueda igual que detenemos cualquier otro comando con Control+C, y redirigir la salida como también hemos aprendido; **find** puede ser muy útil en tus scripts.

Cuando siendo usuarios normales busquemos ficheros fuera de nuestro directorio personal, se nos mostrarán algunos mensajes de error porque habrá directorios a los que no tendremos permiso para entrar a buscar (p.e. `/root`). Estos mensajes de error nos despistarán a la hora de interpretar la salida, así que podemos enviar `stderr` a `/dev/null` (ya sabes, **find ... 2>/dev/null**).

Uso de locate

En primer lugar, ejecutaremos el comando **updatedb** para actualizar la base de datos. Es una buena idea poner este comando en *cron* como veremos posteriormente para que se ejecute automáticamente todos los días a una hora en la que el sistema no tenga mucha carga. Esto nos permitirá agilizar nuestras búsquedas.

A continuación, el uso de **locate** no es nada del otro mundo. Además, es mucho menos flexible que **find** (lo cual es perfectamente comprensible una vez que nos han explicado su forma de funcionar). **locate** sólo sirve para buscar ficheros por nombre, y en todo el sistema de archivos. Ejemplos:

```
# locate so
[... muuucha salida ...]
# locate '*.so'
[... las librerías del sistema ...]
```

La similitud con **find** es que los criterios con wildcards deben ser entrecomillados con comillas simples. Las diferencias son: no nos permite especificar directorio, busca en toda la ruta al fichero (no sólo en el nombre del fichero), y además, si le pasamos un criterio sin wildcards, devolverá todos los nombres de fichero cuya ruta completa contenta ese criterio (nótese en el primer comando de ejemplo), a diferencia de **find** que sólo devolvería los ficheros con ese nombre exacto.

man y las páginas del manual

Ya venimos usando esta útil funcionalidad algún tiempo. Es muy útil disponer de documentación en formato electrónico en el sistema, de esta forma no hay que memorizar las opciones de un comando de memoria.

Existen varias *secciones* en el manual del sistema, cada una de ellas destinada a un tipo de comandos. También, hay páginas que documentan funciones de lenguajes de programación si esta documentación está instalada. Puedes ver cuáles son estas categorías con **man man**; además ahí hay mucha información interesante.

Puede ocurrir que existan dos páginas del manual que se llamen igual y que estén en secciones distintas. **man** sólo mostrará la primera de ellas, así que si queremos una específica, deberíamos indicar su sección (**man SECCION comando**).

Otro aspecto interesante ya comentado es definir la variable de sistema PAGER como `/usr/bin/less` para poder recorrer las páginas del manual con todas las funciones interesantes de **less**. Viendo las páginas del manual con less podemos usar todas sus útiles funcionalidades, ya descritas en la la sección de nombre *Comandos útiles de less* en Capítulo 6.

Algunas distribuciones de Linux no traen instaladas las páginas del manual en castellano por defecto, pero pueden tener ese paquete en los discos de instalación o en Internet, llamado generalmente `manpages-es` o nombres similares.

Apagar y reiniciar la máquina desde el shell

El comando **poweroff** se encuentra en la mayoría de las distribuciones Linux y sirve para apagar la máquina. Generalmente el comando **halt** produce el mismo efecto. **reboot** reinicia la máquina. En algunas distribuciones, solo `root` tiene permiso para usar estos comandos por defecto, así que si se quiere que usuarios no privilegiados puedan apagar o reiniciar el sistema, se debe recurir a *sudo* o hacer SUID estos archivos, ya visto en Capítulo 5

Capítulo 10. Personalización del shell BASH

Introducción

Todos los capítulos hasta ahora, hemos intentado dar un punto de vista general para Linux y también válido para muchos sistemas de tipo Unix y/o que conformen con POSIX. Para ello decidimos hacerlo basándonos solo en la consola de Linux. Queremos hacerle ver a los usuarios que la consola de Linux es MUY MUY potente, y que desde ella podemos hacer (casi) cualquier cosa que se nos ocurra o necesitemos, solo se requiere una base sólida.

Hasta ahora sólo hemos hablado del shell Bash, pero ni si quiera hemos profundizado en ella. En este capítulo se intentarán dar unas nociones sólidas para personalizar el shell Bash que es el estándar en muchos siestemas Unix y en casi todas las distribuciones de Linux.

BASH es una aplicación Unix, y como muchas de estas aplicaciones, su nombre no es más que un acrónimo, y en este caso, BASH significa Bourne Again SHell. Se trata de una shell de Linux compatible con POSIX que implementa parte de los comandos de C shell y toda la sintaxis de Bourne SHell.

Como (casi) toda aplicación, tiene unos ficheros de "configuración", y ciertas "funcionalidades" que no están claras a simple vista. Mostrar algunas de estas cosas es el objetivo de este capítulo.

Variables interesantes. Personalización del Prompt.

Como se ha comentado en el capítulo 7 (Capítulo 7) en BASH podemos utilizar variables que, normalmente, serán todas en mayúsculas. Pues existen ciertas variables que el shell bash utiliza para su funcionamiento tanto interno como de cara al usuario.

Existen tanto variables de lectura, como variables de lectura-escritura. Las variables de lectura "simplemente" nos darán información sobre el estado del shell o del sistema, sin embargo, las de lectura-escritura nos van a permitir cambiar el comportamiento de bash

El shell bash (como casi todos los shells) cuando recibe una orden (por ejemplo, un comando pasado por teclado) lo primero que hace es ver si la orden está dentro de sus órdenes internas (como return, exit...), luego mira en los alias, y después busca el comando en el $PATH (Capítulo 3). Las órdenes (o comandos) internas se explicaron en el capítulo 7 (Capítulo 7), los comandos más comunes se describieron en el capítulo 9 (Capítulo 9) y en este capítulo vamos a tratar los alias.

Pero antes de tratar los alias, vamos a dar una vuelta por las variables más interesantes de bash y por sus ficheros de configuración.

La variable que más llama la atención a cualquier usuario que esté personalizándo el shell bash es la variable `$PS1`. Esta variable describe el prompt que bash mostrará al usuario antes de pedir un comando:

```
usuario@host $ echo $PS1
\u@\h \$
```

Como he dicho antes, las variables que para esto nos interesan son perfectamente modificables, un ejemplo de esto sería:

```
usuario@host $ PS1="\u@\h: \w \$"
```

```
usuario@host: ~ $
```

Ahora vemos cómo ha cambiado el prompt. Como ejemplos de prompts más elaborados se pueden probar:

```
usuario@host $ PS1="{\u@\h[\w] \$::- "
{usuario@host[~] $::- PS1="::-   \t\n::- \u@\h[\w]\$::- "
::-   HH:MM:SS
::- usuario@host[~]$::-
```

En resumen, podemos hacer que nuestro prompt contenga cualquier caracter que nosotros queramos, además de información útil. Los caracteres especiales más relevantes que se cambian por información útil al mostrar el prompt son:

- `\h` es el nombre de host hasta el primer punto.
- `\H` el nombre completo del host.
- `\n` nueva linea.
- `\s` es el nombre del shell, en nuestro caso, "bash"
- `\t` es la hora en formato 24 horas.
- `\u` es el nombre de usuario actual.
- `\v` es el número de versión de bash.
- `\w` es el directorio de trabajo actual.

Dentro de esta variable, PS1, también podemos hacer que se sustituya la salida de cualquier comando igual que hacemos en los scripts. Prueba con `PS1=$(date)`.

Podemos hacer que partes del prompt aparezcan en color. Lo que queramos que aparezca entre colores, lo precederemos por `\[\033[COLORm\]` , teniendo en cuenta que al final del prompt debemos reestablecer el color nulo si no queremos que los comandos que escribamos salgan en color. Esto lo conseguimos con `\[\033[0m\]` . COLOR puede ser una secuencia con formato `A;B` , donde A vale 0 ó 1 y B vale desde 30 a 37. Cada una de las combinaciones posibles resulta en un color distinto. Pruébalas para encontrar el que busques. Por ejemplo:

```
$ PS1='\[\033[01;32m\]\u@\h \[\033[01;34m\]\W \$ \[\033[00m\]'
usuario@host directorio $ # "usuario@host" saldrá en verde y "directorio"
                                    en azul
```

Tabla 10-1. Combinaciones de colores en bash

Combinación	Color
0;30	Negro
0;31	Rojo
0;32	Verde
0;33	Marrón
0;34	Azul
0;35	Púrpura
0;36	Cian

0;37	Gris claro
1;30	Gris oscuro
1;31	Rojo claro
1;32	Verde claro
1;33	Amarillo
1;34	Azul claro
1;35	Púrpura claro
1;36	Cian claro
1;37	Blanco
Colores de fondo	
40	Negro
41	Rojo
42	Verde
43	Marrón
44	Azul
45	Púrpura
46	Turquesa
47	Gris

Los colores de fondo se usan precediendo el valor del color del fondo a los ya vistos para el color de las letras. Por ejemplo, `\[\033[41;1;33m\]` producirá texto amarillo sobre fondo rojo. Hay que tener en cuenta que `\033` y `\e` son equivalentes y pueden usarse indistintamente.

Además estas combinaciones de colores también se usan en otras cosas, como por ejemplo la variable de entorno `LS_COLORS` que contiene con qué color se muestra cada tipo de elemento por defecto.

Hay algunas más combinaciones posibles con colores y demás modificaciones para la variable `PS1`. Un documento muy extenso al respecto es el Bash-Prompt-COMO que se puede encontrar en varios sitios de la red. De ese documento el siguiente script, que sirve para ver las combinaciones disponibles:

```
#!/bin/bash
#
#   This file echoes a bunch of color codes to the
#   terminal to demonstrate what's available.  Each
#   line is the color code of one forground color,
#   out of 17 (default + 16 escapes), followed by a
#   test use of that color on all nine background
#   colors (default + 8 escapes).
#

T='gYw'   # The test text

echo -e "\n                 40m     41m     42m     43m\
     44m     45m     46m     47m";

for FGs in '    m' '   1m' '  30m' '1;30m' '  31m' '1;31m' '  32m' \
           '1;32m' '  33m' '1;33m' '  34m' '1;34m' '  35m' '1;35m' \
           '  36m' '1;36m' '  37m' '1;37m';
    do
     FG=${FGs// /}
```

```
  echo -en " $FGs \033[$FG  $T  "

  for BG in 40m 41m 42m 43m 44m 45m 46m 47m;
  do
   echo -en "$EINS \033[$FG\033[$BG  $T  \033[0m";
  done
  echo;
 done
echo
```

La variable `PS2` contiene el prompt complementario, por ejemplo el prompt que aparece cuando escribimos un bucle directamente en la terminal.

La variable `PROMPT_COMAND` contiene el comando que se ejecutará cada vez inmediatamente antes de que bash muestre el prompt.

La variable `PATH` ya se comentó con anterioridad. Contiene una lista de directorios separados por ":" en los que se buscan comandos. El orden de busqueda es lineal desde el principio, esto es, si hay dos programas con el mismo nombre en directorios distintos y ambos están en el PATH, el programa del directorio que aparezca primero en la variable será el que se ejecute. Ya vimos que esto se podía comprobar con **which <comando>**.

La variable `EDITOR` contiene la ruta al editor por defecto. Lo usan algunos programas además de bash.

Hay muchas variables más cuyo conocimiento no es imprescindible. Puedes ver todas las variables de entorno con el comando **set**. La página del manual de bash las enumera y además contiene muchísima información importante. Es muy conveniente leerla: **man bash**

Alias

Un `alias` nos permite nombrar a un comando como nosotros queramos. Es útil para poner nombres más cortos a comandos largos o bien para no tener que escribir todas las opciones de un comando si las usamos siempre. Por ejemplo, la opción `--color=auto` de **ls** es muy común, así que es bastante frecuente hacer algo como esto:

```
$ alias ls='ls --color=auto'
```

Así, cada vez que escribamos **ls** en bash, éste lo interpretará como **ls --color**. Igual que pusimos `ls` en la parte izquierda de la asignación, podríamos haber puesto cualquier otro nombre que quisiésemos, por ejemplo:

```
$ alias listar='ls -l --color=auto'
```

Y desde esto en adelante usar **listar** como otro comando cualquiera. En los alias caben incluso *listas* de comandos, por ejemplo:

```
$ alias comandos='comando1 && comando2'
```

donde `comando1` y `comando2` son comandos con sus opciones y argumentos si aplican. El **&&** es lo que se conoce en bash como una LISTA. Esto es, se ejecutará `comando1` y si termina con éxito (devuelve 0 a bash) entonces se ejecutará `comando2`. Si un comando en una lista no termina con éxito,

entonces no se ejecutará el resto. Se pueden poner más de dos comandos, por ejemplo: **comando1 && comando2 && comando3**

Si en un momento determinado nos interesa que un alias deje de existir, usaremos **unalias nombre**, donde `nombre` es el nombre que dimos al alias que queremos que deje de existir.

En la página del manual de bash también hay una sección sobre Alias y otra sobre Listas.

Ficheros asociados

Hemos estado viendo cómo personalizar algunos aspectos de bash pero sin embargo hasta ahora los perderíamos al salir de la terminal. Bash lee algunos ficheros en cada inicio. En estos ficheros podremos colocar variables y alias para no tener que cambiarlos cada vez.

Antes necesitamos saber lo que es un *shell de entrada o login* y un *shell interactivo*. Un shell de entrada es el que obtenemos tras hacer login en el sistema en la consola, y un shell interactivo es el que obtenemos, si estamos en modo gráfico y abrimos un *xterm*, o sea, una ventana de terminal.

Para los shells de entrada o de login, primero se lee el fichero `/etc/profile` que contiene definiciones de variables y alias comunes a todos los usuarios del sistema. A continuación se lee el fichero `~/.bash_profile`, en el directorio de cada usuario. En este fichero, cada usuario define sus variables y alias, que sobreescriben a los de `/etc/profile`.

Para los shells interactivos, sólo se lee el fichero `~/.bashrc`, distinto para cada usuario. Puesto que generalmente no se quiere una configuración distinta en un xterm o en una terminal de login, es bastante común poner las definiciones de variables y alias en `~/.bash_profile`, y que `~/.bashrc` se parezca a:

```
if [ -f /etc/profile ]; then
 . /etc/profile
fi

if [ -f /etc/bashrc ]; then
 . /etc/bashrc
fi

source ~/.bash_profile
```

source interpreta el archivo que se le pasa como primer argumento como órdenes de bash. Es prácticamente equivalente al punto **.**

Capítulo 11. Procesos. Señales.

Introducción y Conceptos Básicos sobre Procesos y Tareas

En el capítulo 5 ya hicimos una pequeña incursión al uso de los *procesos* , pero en este otro vamos a hacer una explicación mucho más extensa que nos servirá para manejarnos entre procesos y *tareas*.

En un sistema Linux, que como ya sabemos es *multitarea*, se pueden estar ejecutando distintas acciones a la par, y cada acción es un proceso. Explicaremos en este capítulo como manejar los procesos y aprovechar las peculiaridades de Linux en lo referente a la capacidad multitarea, activando procesos en *primer y sengundo plano*, mostrando listas y *árboles de procesos*, *suspendiéndolos* y luego volviéndolos a activar, *matándolos*, haciendo que sigan ejecutándose una vez se halla cerrado sesión o cambiando la prioridad de las tareas.

Todo lo anterior lo iremos descubriendo en subsecciones sucesivas. De momento nos centraremos en ver lo más básico (conceptualmente hablando) de lo que es un proceso. Cuando ejecutamos un comando en el shell, sus instrucciones se copian en algún sitio de la memoria RAM del sistema para ser ejecutadas. Cuando las instrucciones ya cumplieron su cometido, el programa es borrado de la memoria del sistema, dejándola libre para que más programas se puedan ejecutar a la vez. Cada uno de estos programas que pasan a la memoria del sistema y son ejecutados es lo que conocemos con el nombre de PROCESO. El tiempo que este proceso estuvo en la memoria del sistema ejecutándose, se conoce como TIEMPO DE EJECUCIÓN de un proceso.

La gestión de procesos en Linux no difiere demasiado de la que se hace en cualquier otro sistema UNIX. Así, el encargado de asignar una parte de la memoria a un proceso es el KERNEL de Linux (parte central del Sistema Operativo), quien decide cuánta memoria dar a cada proceso y cómo repartir la capacidad del microprocesador entre los procesos que se están ejecutando. Unos procesos serán más importantes que otros, y de alguna manera tendrán "preferencia" a la hora de pedir cáculos o instrucciones al microprocesador sobre otros procesos. Este concepto se llama PRIORIDAD de un proceso. Por lo tanto, debe quedar claro que es el kernel quien "manda" sobre los procesos y controla su ejecución simultánea y el reparto de los recursos de la máquina.

Como todo en los sistemas UNIX, y por extensión en Linux, los procesos también tienen un dueño. Ese dueño es el usuario que ejecuta el programa que da lugar al proceso. Esto viene a explicar parte de la seguridad de Linux en cuanto a permisos. Como los archivos (y dispositivos como más adelante veremos) tienen un usuario, un grupo y unos permisos determiandos asignados, y también los procesos, un proceso contará de cara al sistema de ficheros con los permisos que tenga el usuario que lo creó. Resumidamente, un proceso, respecto al sistema de ficheros, podrá hacer tanto como el usuario al que pertenece. Un programa ejecutado por un usuario no privilegiado nunca podrá leer en `/root` o borrar o escribir cosas en `/usr/bin`. Un usuario normal no podrá tampoco modificar parámetros de procesos que no le pertenezcan. `root` podrá ejercer cualquier acción sobre cualquier proceso.

Un proceso puede crear a su vez nuevos procesos. En la práctica, todos los procesos son hijos del primer proceso ejecutado durante el arranque, el proceso "init". Así, un proceso hijo, a pesar de ser uno nuevo, guarda una relación con el proceso que lo creó (proceso padre).

Procesos

Propiedades de los procesos

Básicamente, un proceso tiene estas propiedades:

- Un *número identificador*, (Process ID o PID), identificador de proceso. Es necesario para referirnos a un proceso en concreto de los varios en ejecución.
- Un *PPID* (Identificador del proceso padre), es el número que indica qué proceso creó al proceso en cuestión.
- Un *estado*; habrá momentos en los que un proceso seguirá existiendo en el sistema, pero no estará haciendo nada realmente. Puede estar esperando a que una SEÑAL le sea enviada (sobre lo que trataremos más tarde) para volverse activo, o a nosotros como usuarios nos puede interesar detenerlo o pausarlo bajo determinadas circunstancias. Los estados más importanes son dormido (S), y en ejecución (R).
- Un *dueño*, generalmente es el usuario que lo ejecutó, y como ya se dijo, el proceso hereda los permisos del usuario de cara al sistema de ficheros. Si un ejecutable es SUID, al ejecutarse, el proceso resultante adquiere los permisos del usuario dueño del archivo. Generalmente, un usuario normal no puede ejercer control sobre procesos que no le pertenecen.
- Una *prioridad* que determina su importancia. Ya vimos que no todos los procesos eran igual de importantes. Veremos cómo cambiar la prioridad de un proceso más adelante.

Mostrando los procesos en ejecución y sus propiedades.

Para empezar, podemos ver las tareas y sus subtareas en una estructura anidada mediante el comando **pstree**. Al ejecutarlo, veremos algo como:

```
# pstree

init-+-5*[agetty]
     |-bdflush
     |-cron
     |-devfsd
     |-eth0
     |-gdm---gdm-+-X
     |           `-gdmgreeter
     |-keventd
     |-khubd
     |-kreiserfsd
     |-kscand
     |-ksoftirqd_CPU0
     |-kswapd
     |-kupdated
     |-login---bash---pstree
     `-syslog-ng
```

Asimismo, **pstree -p** muestra entre paréntesis el número identificador (*PID*) de los procesos, algo muy importante cuando queremos pasar de actuar de forma pasiva a interactuar con los procesos, cosa que normalmente haremos señalando sus PIDs.

Aunque esta información estructurada resulta interesante, existen dos programas muy conocidos - cada cual más avanzado - que muestran una cantidad ingente de información sobre los procesos, estos son **ps** y **top**.

ps muestra una lista de los procesos en ejecución. Prueba **ps u** , que muestra los procesos que pertenecen al usuario actual. La combinación más habitual de opciones es **ps aux** , que muestra información detallada de todos los procesos en ejecución. Algunos de los campos más importantes mostrados por **ps** son:

- *USER* - usuario dueño del proceso.
- *PID* - número identificador del proceso.
- *%CPU* - porcentaje de uso del microprocesador por parte de este proceso.
- *%MEM* - porcentaje de la memoria principal usada por el proceso.
- *VSZ* - tamaño virtual del proceso (lo que ocuparía en la memoria principal si todo él estuviera cargado, pero en la práctica en la memoria principal sólo se mantiene la parte que necesita procesarse en el momento).
- *RSS* - tamaño del proceso en la memoria principal del sistema (generalmente son KBytes, cuando no lo sea, se indicará con una M detrás del tamaño).
- *TTY* - número de terminal (consola) desde el que el proceso fue lanzado. Si no aparece, probablemente se ejecutó durante el arranque del sistema.
- *STAT* - estado del proceso.
- *START* - cuándo fue iniciado el proceso.
- *TIME* - el tiempo de CPU (procesador) que ha usado el proceso.
- *COMMAND* - el comando que inició el proceso.

La página del manual de Linux sobre **ps** es especialmente útil. Informa sobre otras opciones y propiedades de los procesos en detalle.

Al fin y al cabo, lo que **ps** nos muestra es una "instantánea" de los procesos del sistema en el momento que ejecutamos el comando, así que a priori no podremos ver la evolución.

top es la versión interactiva de **ps**, y tiene algunas utilidades interesantes añadidas. Si lo ejecutamos en una terminal y sin opciones, aparecerá arriba información del sistema: usuarios, hora, información del tiempo de funcionamiento de la máquina, número de procesos, uso de CPU, uso de memoria (nótense las considearciones que hicimos previamente sobre la gestión de memoria en Linux), y uso del swap. A continuación tenemos una lista de procesos similar a la que nos mostraba **ps** con la diferencia de que esta se actualiza periódicamente, permitiéndonos ver la evolución del estado de los procesos.

Con **top**, tenemos dos posibilidades de especificar opciones, bien en línea de comandos en el shell, o bien interactivamente (mientras está en ejecución y sin salir de él). La página del manual de **top** es también muy buena, con descripciones detalladas de los campos y de las opciones, tanto de línea de comandos como interactivas. Léela y prueba las distintas opciones.

```
# top

 22:22:19 up 1 day,  9:41,  4 users,  load average: 0.13, 0.06, 0.02
 78 processes: 77 sleeping, 1 running, 0 zombie, 0 stopped
 CPU states:   1.4% user,   3.6% system,   0.0% nice,  95.0% idle
 Mem:    256192K total,   251776K used,     4416K free,    31684K buffers
 Swap:   409648K total,     7112K used,   402536K free,   115516K cached

  PID USER PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
```

```
 414   root    14  -10 59416  25M  8772 S<   0.9 10.3   3:12 XFree86
2131   usuario 19    0  1052 1052   820 R     0.7  0.4   0:00 top
2129   usuario 14    0  6968 6968  4364 S     0.5  2.7   0:00 xmms
 618   usuario 11    0  3988 3988  2408 S     0.3  1.5   0:15 sawfish
2130   usuario 11    0  6968 6968  4364 S     0.3  2.7   0:00 xmms
   6   root    10    0     0    0     0 SW    0.1  0.0   1:31 kupdated
2085   usuario 10    0  6968 6968  4364 S     0.1  2.7   0:12 xmms
   1   root     9    0   472  432   412 S     0.0  0.1   0:06 init
   2   root     9    0     0    0     0 SW    0.0  0.0   0:00 keventd
   3   root    19   19     0    0     0 SWN   0.0  0.0   0:07 ksoftirqd_CPU0
   4   root     9    0     0    0     0 SW    0.0  0.0   0:10 kswapd
   5   root     9    0     0    0     0 SW    0.0  0.0   0:19 bdflush
  50   root     9    0     0    0     0 SW    0.0  0.0   0:00 khubd
  84   root     9    0     0    0     0 SW    0.0  0.0   0:00 kreiserfsd
 109   root     9    0     0    0     0 SW    0.0  0.0   0:00 eth1
 190   root     9    0   608  604   488 S     0.0  0.2   0:01 syslogd
 193   root     9    0  1248 1240   408 S     0.0  0.4   0:00 klogd
 196   root     9    0  2816 2776  1728 S     0.0  1.0   0:00 named
 199   root     9    0  2816 2776  1728 S     0.0  1.0   0:01 named
```

Tareas de Bash. Programas en primer y segundo plano.

El control de tareas es una interesante característica que bash nos ofrece. Así, cada programa o tubería que se ejecute tiene asignado un número de tarea por parte de bash (diferente al PID).

Los programas que podemos ejecutar desde una línea de comandos pueden estar en primer o segundo plano, y de hecho pueden pasar de un estado a otro. Una consola concreta está bloqueada cuando un proceso está en primer plano, mientras que si está en segundo plano la consola está libre y podemos teclear comandos en ella.

Esto está más relacionado con la terminal en sí (bash en nuestro caso) que con el concepto de proceso que hemos aprendido. Normalmente al ejecutar un comando no recuperamos el prompt hasta que éste no termina. La necesidad que nos surge es poder usar esa terminal sin abrir otra nueva a la vez que nuestro comando sigue haciendo lo que tenga que hacer.

En bash, ponemos una tarea en segundo plano añadiendo un `&` al final del comando. El comando se ejecuta mientras la terminal queda libre.

```
$ sleep 10 &
[1] 6190
$ # tras 10 segundos presionamos INTRO
[1]+  Done                    sleep 10
```

El comando **sleep** simplemente espera el número de segundos del primer argumento. Al ejecutarlo en segundo plano, bash imprime la línea `[1] 6190` y nos devuelve al prompt, aunque el comando se sigue ejecutando ahora en segundo plano. El número entre corchetes es el NÚMERO DE TAREA (o, también llamado trabajo o *job*) que bash le asignó al pasarlo a segundo plano, y el número que hay a continuación (6190) es el número de proceso que ha generado la ejecución del comando. Cada vez

que al presionar INTRO hay novedades en la gestión de tareas, bash nos informa de ello. En este caso, ha terminado la tarea [1].

Teniendo una tarea en segundo plano, arrojará su salida a la terminal actual a pesar de todo. Podemos usar las redirecciones que ya conocemos para evitarlo.

Para recuperar una tarea en segundo plano a primer plano, podemos usar **fg**, seguido de %N , donde N es el número de tarea que queremos recuperar, por ejemplo **fg %2** traería a primer plano la segunda tarea.

Con una tarea en primer plano, podemos usar `Ctrl+Z` y la detendremos. Ahora podemos elegir entre continuar con esta tarea en primer plano, o mandarla a segundo plano. Para continuarla en primer plano, tecleraríamos **fg %1**. Para seguirla en segundo plano, usaremos **bg %1**, suponiendo que es la tarea número 1.

Un comando ejecutándose en primer plano puede ser cancelado con `Ctrl+C`, lo que a su vez hará desaparecer al proceso resultante de la tabla de procesos del kernel.

Señales.

Las SEÑALES son la forma que tienen los procesos de comunicarse entre sí, y el kernel de comunicarse con ellos. Existe un número determinado de señales que se pueden enviar a un proceso, y cada una de ellas tiene sobre él una acción distinta: pausarlo, reanudarlo, cancelarlo...

Conceptualmente, debemos entender, que un proceso recibe una señal, y tiene predeterminada una forma de actuar ante esa señal. Así, hay un conjunto de señales ante las cuales todos los procesos se comportan igual, sin embargo, hay señales ante las que procesos distintos se comportarán de manera distinta, e incluso habrá procesos que ignoren determinadas señales. Habrá otras que no podrán ignorar (la de detenerse, por ejemplo) porque son comunes a todos los procesos.

Las señales se denotan por SIG*NOMBRE*, donde NOMBRE es el nombre que se le da a la señal. Además, las señales más usadas tienen un *número de señal* asociado. Veamos las más comunes:

- `SIGHUP` (num 1) Causa que un proceso "relea" sus archivos de configuración. Todos los programas no implementan esta señal.
- `SIGINT` (num 2) Interrumpir (cancelar) un programa. Es la señal que se le envía a un proceso cuando presionamos Ctrl+C en teclado con un proceso en primer plano.
- `SIGKILL` (num 9) "Matar" un proceso. Los programas no pueden decidir cómo responder a esta señal, terminan inmediatamente cuando la reciben. Sólo debería ser usada en el caso de necesitar terminar un proceso que ha dejado de responder y no funciona adecuadamente.
- `SIGUSR1` (num 10) Señal de usuario. Cada programa puede decidir cómo responder a esta señal.
- `SIGSEGV` (num 11) Señal de violación de segmento (segmentation fault). Esta señal le es enviada a un proceso cuando intenta acceder a memoria que "no es suya", que está fuera de la zona de memoria que ese proceso puede usar. El proceso es detenido inmediatamente.
- `SIGPIPE` (num 13) Tubería no existente. Esta señal se envía a un proceso que estaba escribiendo a una tubería o pipe cuando el destino de la tubería ya no existe.
- `SIGTERM` (num 15) Terminar. Se envía esta señal a un proceso cuando queremos que termine normalmente, haciendo lo que necesite antes de terminar.
- `SIGCHLD` (num 17) Hijo terminó. Esta señal se envía a un proceso padre automáticamente cada vez que un proceso hijo termina.

Se puede ver información más detallada sobre las señales con **man signal**.

El comando **kill -SEÑAL PID** es el que se usa para enviar señales a los procesos. Podemos indicar varios PIDs para que reciban esa señal en un solo comando separándolos con espacios al final. Son equivalentes:

```
# kill -SIGKILL 1283
# kill -9 1283
```

Mirar en cada momento qué PID tiene el proceso originado por un comando puede ser incómodo. Por eso existe el comando **killall -SEÑAL nombre_comando**, donde "nombre_comando" es el nombre del comando que dió lugar al proceso en cuestión.

La página del manual de **kill** es importante, conviene consultarla.

Prioridad de los procesos. El comando nice

Lo más común es tener muchos procesos a la vez ejecutándose en nuestra máquina. Pero no todos son igual de importantes. Por ejemplo, si estamos grabando un CD, este debería ser un proceso más importante que el resto, porque si el disco duro no envía los datos a la velocidad suficiente a la grabadora, perderemos nuestro disco grabable. Así que le diremos al kernel: "si el procesador no puede con todo, lo último que debes retrasar es la grabación del CD". Podríamos notar que el resto de aplicaciones van más lentas, pero eso nos garantizaría en una mayor medida que la grabación no va a ser interrumpida.

La forma de conseguir esto es, ejecutando desde el principio el comando interesado mediante **nice**, o bien, conseguir su PID si ya se está ejecutando y usar **renice** para cambiar su prioridad.

Algunos ejemplos:

```
$  nice -n PRIORIDAD COMANDO
$  renice PRIORIDAD PID_PROCESO
```

`PRIORIDAD` es un valor que va desde -20 a +20 (con el signo incluido). -20 es la prioridad más alta (al contrario de lo que cabría pensar), y +20 la más baja. Sólo root puede establecer un proceso a una prioridad negativa, los usuarios como máximo pueden poner un proceso en prioridad 0.

`COMANDO` es el comando que queremos ejecutar (sólo aplicable con **nice**), el mismo comando que ejecutaríamos normalmente con todos sus parámetros y opciones.

`PROCESO` (sólo aplicable con **renice**) cambia la prioridad del proceso cuyo PID es PID_PROCESO al nuevo valor indicado.

Capítulo 12. Utilidades de compresión y empaquetado de ficheros y directorios.

Introducción

Hasta ahora hemos visto como manejarnos con varios comandos, con editores de texto, con la shell y con el sistema en general. Sin embargo, no se ha entrado en un tema bastante común en el mundo de la informática: la compresión de datos.

En este capítulo se pretenden enseñar las utilidades más comunes de compresión y archivado de ficheros y directorios de Linux. Además, estas son comunes a muchos, por no decir todos, los Unix.

Visión general del problema y su solución

Antes de empezar con listados de utilidades, rutas, binarios, ficheros y nuevos comandos que pueden asustar a cualquiera vamos a presentar de una forma general el problema que queremos resolver.

Suponemos que tenemos ciertos datos que queremos guardar, empaquetar o comprimir para llevarlos a una unidad de backup, para enviarlos por correo electrónico, para adjuntarlos a un trabajo o simplemente para tener una copia de seguridad de algo sobre lo que vamos a trabajar.

En el mundo Unix predominan dos algoritmos de compresión de datos distintos; estos son *gzip* y *bzip2*. Ambos son muy eficientes y no podemos decir que uno sea mejor que otro. Presentan grandes diferencias internas pero a nivel de usuario lo que podemos decir es que bzip2 comprime más pero consume definitivamente más recursos que gzip para comprimir/descomprimir el mismo fichero.

Históricamente en el mundo Unix se ha usado mucho la utilidad **tar** para enviar backups a dispositivos de cinta. **tar** se encarga de unir un conjunto de ficheros en uno solo. En principio **tar** no provee compresión alguna ya que lo "único" que hace es poner un fichero a continuación del anterior poniendo en medio unos *separadores*. Aunque la explicación no es muy rigurosa, nos vale para entender el por qué de **tar** hoy en día.

Los algoritmos de compresión que hemos citado antes (bzip2 y gzip) solo son capaces de operar sobre un fichero; es decir, toman un fichero de entrada y generan un fichero comprimido. No pueden tomar como entrada un directorio o varios ficheros a la vez. Por eso es importante **tar**. Con la ayuda de **tar** podemos convertir todo un directorio en un solo fichero y una vez tenemos ese fichero generado, aplicarle uno de los algoritmos de compresión.

Utilidades para la línea de comandos

Cualquier distribución moderna de Linux incluirá tanto **tar** como los paquetes con las librerías y utilidades para comprimir y descomprimir ficheros tanto con gzip como con bzip2. Además, actualmente **tar** está muy integrado con ambos algoritmos de compresión y nos evita tener que utilizar comandos algo más complicados para comprimir y descomprimir ficheros

Los comandos que usaremos en esta sección son: **tar**, **gunzip**, **gzip**, **bzip2** y **bunzip2**. Como podemos ver son nombres muy intuitivos. Ya tenemos todo listo para empezar a comprimir y descomprimir ficheros y directorios. Lo primero que haremos será elegir un fichero a comprimir; por ejemplo, `/etc/passwd`.

Comprimir y descomprimir un solo fichero

Para comprimir el fichero que hemos escogido con gzip haremos:

```
$ gzip -c /etc/passwd > passwd.gz
```

Esto habrá dejado un fichero llamado `passwd.gz` en el directorio donde ejecutamos el comando. Ahora podemos probar a descomprimirlo y a comprobar si todo fue bien

```
$ gunzip passwd.gz
$ diff /etc/passwd passwd
$
```

Es muy importante ver que el comando **gunzip passwd.gz** ha borrado del directorio el fichero `passwd.gz`.

Para hacer algo parecido con bzip2, podemos hacerlo así:

```
$ bzip2 -c /etc/passwd > passwd.bz2
$ bzip2 passwd.bz2
$ diff /etc/passwd passwd
$
```

Comprimir y descomprimir directorios completos

Como ya hemos comentado antes, ninguno de los dos algoritmos de compresión que veremos soporta trabajar sobre varios ficheros a la vez; si no que su entrada será un único fichero sobre el que se aplicará la compresión. Si queremos comprimir directorios enteros lo que tenemos que hacer es utilizar **tar** para crear un solo fichero a partir de varios a base de concatenarlos.

Para crear un único fichero a partir de un directorio podemos hacer lo siguiente:

```
$ tar cf directorio.tar directorio/
```

Obtendremos en el directorio un fichero `directorio.tar` que contiene todo el arbol de directorios que hubiera colgando de `directorio/`. Si queremos comprimir este fichero lo podemos hacer como hemos visto antes:

```
$ gzip directorio.tar
```

Y tendremos un fichero llamado `directorio.tar.gz` que contendrá todo el arbol de directorios que originalmente colgaba de `directorio/`. Lo mismo podríamos haber hecho utilizando el comando **bzip2** en lugar de **gzip** si queríamos utilizar el algoritmo bzip2.

Para desempaquetar el fichero `directorio.tar` lo podemos hacer con el siguiente comando:

```
$ tar xf directorio.tar
```

Todo junto

Como ya hemos comentado antes, actualmente **tar** está muy integrado tanto con gzip como con bzip y podemos comprimir y descomprimir directorios enteros con solo un comando.

Además de enseñar cómo comprimir de forma simple, tambien voy a mostrar cómo bzip2 comprime normalmente más que gzip a costa de usar más recursos y tiempo. Como prueba utilizaré el directorio que contiene el código fuente de este manual ya que así, el lector podrá repetir estos comandos y obtendrá resultados muy similares.

Para comprimir un directorio con **tar** lo hacemos así:

```
$ tar zcf manual-cvs.tar.gz manual ❶
$ tar jcf manual-cvs.tar.bz2 manual ❷
```

❶ Se usa ’z’ para gzip

❷ y ’j’ para bzip2

Para ver la diferencia de tamaños podemos usar **ls -l**

```
$ ls -l manual-cvs.tar.*
-rw-r--r--    1 ferdy    ferdy      331236 sep 18 03:11 manual-cvs.tar.bz2
-rw-r--r--    1 ferdy    ferdy      477033 sep 18 03:13 manual-cvs.tar.gz
```

Como podemos ver, el fichero comprimido con bzip2 ocupa bastante menos que el mismo comprimido con gzip; sin embargo, quizá el lector pudo notar que la operación con bzip2 tardó más tiempo en completarse.

Por último veremos cómo descomprimir los ficheros que tenemos:

```
$ tar zxf manual-cvs.tar.gz ❶
$ tar jxf manual-cvs.tar.bz2 ❷
```

❶ Se usa ’z’ para gzip

❷ y ’j’ para bzip2

Si solo queremos extraer un fichero o directorio del fichero comprimido lo podemos hacer indicando el nombre justo al final del comando; por ejemplo:

```
$ tar jxf manual-cvs.tar.bz2 manual.xml
$ ls manual.xml
manual.xml
```

Descomprimiendo otros formatos

Es muy típico querer comprimir/descomprimir otros formatos que son muy utilizados en el mundo Windows como los zip o los rar. Lo primero que hay que decir es que Linux SI puede descomprimir estos formatos; pero las utilidades que lo hacen no son libres.

Para descomprimir un fichero zip vamos a utilizar el comando **unzip** y para descomprimir los ficheros rar utilizaremos el **unrar**.

```
$ unzip fichero.zip
$ unrar fichero.rar
```

Como ya he dicho antes, estas utilidades *no son libres* son gratuitas para uso personal; pero no tenemos acceso al código fuente. Así que deberíamos evitar utilizar estos formatos para comprimir nuestros ficheros. Ya que además de no ser formatos libres, la compresión es *mucho* menor que con los formatos libres que hemos visto antes.

Capítulo 13. Expresiones regulares y sed.

Introducción

En este capítulo vamos a tratar una herramienta muy poderosa, las expresiones regulares y su uso con **sed**. Ahora quizá todo esto suene a chino, la idea de este capítulo no es llegar a convertirse en un experto en expresiones regulares y/o **sed**, ya que se escriben libros exclusivos sobre expresiones regulares y sobre **sed**; pero, una vez acabado, se tendrá el conocimiento suficiente para escribir y leer expresiones regulares que nos facilitarán muchas tareas.

Aunque **sed** y las expresiones regulares están íntimamente ligados, no dependen el uno del otro. Podemos ejecutar scripts y comandos **sed** sin necesidad de tener conocimiento de expresiones regulares, y viceversa. Podemos ejecutar y utilizar expresiones regulares sin necesidad de invocar a **sed** (por ejemplo desde perl).

Dada esta independencia, las expresiones regulares son más una idea que una implementación. Las expresiones regulares, como ya veremos más adelante, son patrones que podremos utilizar para buscar, sustituir... cadenas de texto. Evidentemente, hay mas implementaciones de las expresiones regulares (POSIX, Perl, ...) pero aquí las que explicaremos estarán orientadas a su uso con **sed**.

Pero, ¿qué es sed?

El nombre de **sed**, al igual que otros comandos Unix (y por tanto también de Linux), es un acrónimo que significa Stream EDitor; es decir, un editor de flujos. **sed** nos va a permitir editar flujos de datos pasados, por ejemplo, a través de una tubería (pipe, |). Para más información ver la sección de nombre *Pipes o tuberías* en Capítulo 6.

sed es muy util para hacer modificaciones en flujos de datos o en ficheros de texto (por ejemplo). Estas modificaciones pueden ser añadir o borrar una línea, o un conjunto de ellas, buscar y reemplazar patrones...

Por ejemplo, es muy típico en alguien que haga webs querer sustituir todas las etiquetas   por   a excepción de aquellas que se encuentren (por ejemplo) detrás de un punto. Siempre podríamos abrir nuestro editor favorito e ir haciendo búsquedas y reemplazos como locos. Sin embargo, **sed** apoyándose en las expresiones regulares es capaz de simplificarnos MUCHO la tarea.

Y, ¿qué son las expresiones regulares?

Aunque no deberíamos tomarlo como una verdad absoluta, podemos pensar en las expresiones regulares como una herramienta que nos permite buscar cadenas dentro de otras cadenas más grandes.

Aunque la explicación del párrafo anterior parezca simple y burda a primera vista, realmente no lo es tanto. El hecho de buscar cadenas es algo que se da a diario en muchas aplicaciones. No siempre queremos buscar cadenas por el mero hecho de saber si están ahí o no; otras veces queremos, además, hacer alguna modificación en esas cadenas. O, por ejemplo, podemos querer partir una cadena grande en otras más pequeñas dado un *patrón*.

Pues bien, las expresiones regulares no son más que eso, *patrones* que podremos buscar en cadenas de texo más grandes y largas.

Ya conocemos lo que son las expresiones regulares a nivel conceptual, pero, evidentemente esto se traduce al final en un conjunto de signos con una sintáxis y una grámatica. Hay varios tipos de expresiones regulares, pero en este capítulo solo veremos las expresiones regulares para su utilización con **sed**. Este tipo de expresiones regulares es muy similar a las expresiones regulares de Perl, pero no son 'exactamente' iguales.

Muy bonito, pero, ¿cómo funciona todo esto?

Aunque **sed** se puede utilizar sin necesidad de tener conocimientos sobre expresiones regulares, uno de sus usos más comunes es utilizar expresiones regulares para hacer sustituciones y búsquedas en ficheros y flujos de texto.

sed es capaz de ejecutar un 'pequeño' pero poderoso lenguaje de comandos que nos permitirá manejar los flujos apoyándonos en dos buffers principales.

Primeros pasos con sed

Para dar nuestros primeros pasos con **sed** vamos a hacer una simple sustitución de texto. Vamos a sustituir en un fichero todas las ocurrencias de una palabra, por ejemplo 'zonasiete' por 'ZonaSiete.ORG'.

> **Nota:** En la sección de nombre *Primeros pasos con sed* al principio se verá la sintaxis para utilizar expresiones regulares muy simples; al final se verán los comandos de sed y cómo utilizar ambas cosas (expresiones regulares y comandos) juntas.

La sintaxis de sed *normalmente* es:

```
sed s/patrón/reemplazo/opción
```

Así que para hacer nuestro reemplazo podemos hacer:

```
$ cat fichero | sed s/zonasiete/ZonaSiete.ORG/g > fichero.reemplazado
```

Ahora en `fichero.reemplazado` tendremos el resultado del comando.

Sed tambien dispone de algunos comandos como el que sin darnos cuenta hemos aprendido, el comando de reemplazo (`s`). Los otros dos comandos que aprenderemos son el comando de imprimir (`p`) y el de borrar (`d`).

Los comandos de sed los aplicaremos sobre un grupo de líneas. Ese grupo lo podemos definir con límites fijos (4,6) o utilizando expresiones regulares.

> **Nota:** Las expresiones regulares se estudian en la siguiente sección. Es normal que las siguientes líneas no se entiendan completamente; de todas formas una vez se entiendan las expresiones regulares, será interesante volver a este punto para intentar entender estos ejemplos.

Si por ejemplo queremos eliminar las líneas vacías de un fichero podemos usar algo como:

```
$ cat fichero | sed '/^[ ]*$/d' > fichero.reemplazado
```

Y como ya es costumbre, en `fichero.reemplazado` tendremos el resultado de la operación. Hemos visto cómo definir, con una expresión regular, cómo elegir las líneas sobre las que actuará sed.

Como ejemplo para terminar de demostrar esto será extraer todos los 'Subject' de un fichero mbox:

```
$ sed '/^Subject:.*/s/^[^:]*: \(.*\)$/\1/p;d' < mbox
Subject 1
Subject 2
Otro asunto
Re: Otro asunto
...
```

Al añadir ;d lo que hacemos es borrar las líneas que no concuerdan con la expresión regular. Si no lo pusieramos, veríamos todas las líneas en la pantalla.

Conociendo a las expresiones regulares

Existen muchos títulos que cubren extensamente las expresiones regulares; de hecho existen libros especialmente 'gordos' que *solo* tratan las expresiones regulares. Aquí daremos una pequeña introducción para hacer unas pequeñas búsquedas y sustituciones.

Las expresiones regulares son una herramienta para definir patrones de búsqueda y reemplazo. En ellas se definen lo que llamaremos *átomos* que serán las partes que buscaremos. Se supone que lo que buscamos no lo conocemos (si no no lo buscaríamos), así que tenemos que definir un átomo para cada parte que busquemos. En los patrones podemos hacer referencias a 'un caracter cualquiera', 'el comienzo de línea', 'el final de línea' y cosas así. Para eso se utilizan *caracteres reservados* tambien conocidos como *wildcards*.

Antes de mostrar una tabla con todos los *wildcards* vamos a ver un pequeño ejemplo sobre como usar dos *wildcards* muy comunes, el de 'principio de línea' (^) y 'final de línea' ($).

Por ejemplo para añadir un texto al comienzo de todas las líneas podemos hacer:

```
$ cat loquesea | sed 's/^/Texto al comienzo:/g' > loquesea.nuevo
```

Lo que hace es reemplazar el comienzo de cada línea con ese texto. Hemos entrecomillado la expresión para no tener problemas con la shell. Es una buena costumbre el entrecomillar las expresiones de **sed**.

Veamos el mismo ejemplo pero al contrario. Queremos añadir algún texto al final de todas las líneas de un flujo de datos; lo haremos con la siguiente expresión de **sed**.

```
$ cat unfichero | sed 's/$/Esto se añade al final/g' > unfichero.nuevo
```

Antes de seguir, decir que **sed** acepta los dos tipos de sintaxis que hay en el siguiente ejemplo:

```
sed s/patrón/reemplazo/opción
sed s:patrón:reemplazo:opción
```

> **Nota:** A partir de ahora los ejemplos utilizarán indistintamente una de la otra. De todas formas la segunda forma es más simple de leer cuando hay que 'escapar' caracteres.

Quiero demorar un par de párrafos y ejemplos la llegada de la temida tabla de *wildcards*. Imaginemos que queremos borrar cualquier signo $ al principio de una línea por una x.

Nuestro primer intento puede ser del tipo:

```
sed 's:^$:x:g'
```

Pero si lo pensamos bien, estamos diciendo algo como: 'Reemplaza todas las líneas que tengan justo despúes del comienzo, el final, por una x'. Lo que viene a ser algo como: 'Reemplaza las líneas vacías con una x'. De hecho podemos escribir unas líneas tras correr ese comando y veremos que realmente no hemos conseguido lo que queríamos (reemplazar los $ iniciales por x).

```
$ sed 's:^$:x:g'
$a ver si funciona
$a ver si funciona

x
```

Como vemos, la línea en blanco la ha reemplazado por una x. lo que significa que tenemos que hacerle ver de alguna forma a **sed** que queremos que trate $ como un literal y no como el fin de línea. Esto lo vamos a conseguir añadiendo una '\' al caracter: \$.

Así que escribiríamos nuestra pequeña expresión regular:

```
$ sed 's:^\$:x:g'

$ahora no reemplazó la línea vacía
xahora no reemplazó la línea vacía
```

A partir de ahora cualquier caracter que queramos que **sed** trate como un literal en lugar de un *wildcard*, le añadiremos una '\' delante y diremos que lo hemos *escapado*. Quizá el lector haya notado el siguiente problema: '¿Cómo decirle a **sed** que una \ es un literal y no es el caracter de escape?'. Pues tan simple como que hay que escapar la \ quedando: \\.

Imaginemos que queremos quitar la \ del comienzo de una línea. Escribiríamos nuestra expresión regular así:

```
sed 's:^\\::g'
```

Para terminar de conocer a las expresiones regulares vamos a presentar la tabla de *wildcards*. Todos (o casi) los caracteres y estructuras descritas en la tabla serán explicadas así que no debe asustarse el lector.

Tabla 13-1. Lista de wildcards para expresiones regulares

Caracter/Estructura	Descripción	Ejemplo(s)

Caracter/Estructura	Descripción	Ejemplo(s)
`[ ]`	Los corchetes se utilizan para definir rangos	[0-9] o [a-zA-Z] o uno más completo [A-Za-z0-9,.-_]
`\{ \}`	Las llaves se utilizan para definir el número de veces que ha de repetirse un patrón (o atómo).	ELEMENTO\{3,7\} hará que `ELEMENTO` tenga que ocurrir entre 3 y 7 veces. Y ELEMENTO\{4\} hará qe `ELEMENTO` tenga que ocurrir exactamente 4 veces.
`\( \)`	Nos permite crear 'átomos'. Todo lo que esté entre paréntesis será considerado un 'átomo'.	\([0-9]*Hola\) Buscará una posible secuencia de números precediendo a la palabra *Hola*.
*	Indica que un átomo o elemento podrá o no existir y en caso de existir podrá ocurrir cualquier número de veces.	ELEMENTO*
.	Equivale a cualquier caracter (uno solo).	s:.::g Borrará toda la entrada. s:.*:A:g Cambiará cada línea por 'A'. s:.:A:g Cambiará cada caracter de la entrada por una 'A'.
^	Simboliza el principio de línea.	^$ simboliza una línea vacía.
`[^ ]`	Crea un rango que contiene cualquier caracter menos los especificados entre `[^` y `]`.	[^xyz,] crea un rango que contiene a todos los caracteres menos a `x`, a `y`, a `z` y a `,`.
$	Simboliza el final de línea.	\.$ concuerda con las líneas que acaben en `..`

Para extraer el contenido literal de uno de los átomos utilizaremos la sintaxis: \n donde `n` es el número del átomo que queremos extraer.

Ejemplos más elaborados y divertidos

Una vez conocemos bien a las expresiones regulares no debemos tener miedo a empezar a escribir las nuestras. Sin embargo, las primeras pueden hacerse muy duras, así que para que el lector se sienta a gusto creando expresiones regulares, hemos recopilado algunos ejemplos elaborados e incluso *divertidos* para poner en práctica las expresiones regulares.

Algunos de estos ejemplos se presentan como ejercicios para el lector. Lo bueno (o malo) de las expresiones regulares es que hay varias formas de llegar al mismo sitio. Si se da con una solución distinta a la propuesta, MÁNDANOSLA y la incluiremos en cuanto podamos. (Para ponerte en contacto con nosotros: la sección de nombre *Editores en activo* en Apéndice A).

Antes de meternos con los ejercicios, vamos a ver algunos pequeños ejemplos basados en ficheros comunes del sistema que permitirán al lector familiarizarse con las expresiones regulares y su 'jerga'.

Sed realmente es un minilenguaje de programación capaz de hacer sustituciones realmente complejas de forma muy eficiente. Para ir demostrando su uso utilizaremos un pequeño ejemplo incremental

para sacar la lista de 'targets' de un Makefile. Como ejemplo utilizaremos el Makefile que se provee con las fuentes de este manual.

Lo mejor para resolver los problemas de expresiones regulares es seguir un esquema parecido al siguiente:

Método de resolución de problemas de expresiones regulares

1. Visualizar la entrada
2. Formalizar (¡en lenguaje natural!) la solución (repítela en alto, suele ayudar)
3. Ajustar la formalización a las características vistas
4. Generalizar (este puede ser el paso más complicado) e implementar

No es una regla de oro, ni un método infalible, pero suele funcionar.

Como ya hemos comentado, la idea es extraer solo los targets del Makefile que hay en las fuentes de este mismo manual. Lo primero es formalizar la expresión de una forma general e ir por pasos. Algo así servirá: `Seleccionar las líneas que comienzan con un grupo indefinido de caracteres distintos de espacio seguido de : y de ellas extraer el grupo indefinido de caracteres distintos de : seguido de : y texto libre hasta el final de línea que hay al comienzo de cada línea. Imprimir esa línea, borrar las demás.`

Empezamos por seleccionar las líneas que nos interesan, eso lo podemos hacer con una expresión regular tal que así: /^[^]*:/.

Sobre esas líneas debemos ejecutar el comando de reemplazo (`s`) e implementar la expresión regular que habíamos formalizado. Lo hacemos concatenando el comando a la expresión anterior: /^[^]*:/s/^\([^:]*\):.*$/\1/.

Una vez tenemos el comando de reemplazo debemos ejecutar sobre esas líneas el comando para imprimirlas (`p`) y borrar el resto (`d`) esto lo podemos hacer agregando: p;d a nuestro 'script' en **sed** y ponerlo todo junto:

```
sed '/^[^ ]*:/s/^\([^:]*\):.*$/\1/p;d'
```

Si lo ejecutamos veremos algo como:

```
$ sed '/^[^ ]*:/s/^\([^:]*\):.*$/\1/p;d' < Makefile
.PHONY
all
single
multi
ps
pdf
comprimidos
check
txt
z7
clean
htmlclean
images
fo
```

El target `.PHONY` es un target especial así que nos interesa deshacernos de él. Lo haremos incluyendo una pequeña modificación:

```
sed '/^[^ .]*:/s/^\([^:]*\):.*$/\1/p;d' < Makefile
```

Los siguientes ejercicios se dejan como trabajo al lector:

> **Importante:** Es importante evitar la tentación de mirar las soluciones. De todas formas justo debajo de los ejercicios está escrita la solución 'formal' que hemos implementado; y en una nota al pie, la solución. La solución formal nos puede servir como pista para ponernos a implementar la expresión regular y, además, para inspirarnos en próximas expresiones regulares. De cualquier modo, es mejor intentar no mirar ambas.

Extraer el PID de una línea de syslog. Se pide extraer el número de proceso de una línea del log del demonio syslog. Una línea de dicho log tiene la siguiente forma:

```
MES DIA HORA hostname proceso[PID]: MENSAJE
```

La solución es facil: `Un conjunto indefinido y libre de caracteres, un [, un grupo de dígitos del 0 al 9 de longitud indefinida, un ], un : y texto libre hasta el final de la línea. Extraer el PRIMER GRUPO.` [1]

Extraer el nombre de proceso de una línea de syslog. Continuando con la línea de syslog, ahora se requiere extraer el nombre del proceso.

Parecida a la anterior: `Un grupo de tres letras mayúsculas o minúsculas, un espacio, un grupo de uno o dos dígitos, un espacio, dos grupos de cualquier número de caracteres distintos de espacio, un espacio, un grupo de cualquier número de caracteres distintos de [, texto libre. Extraer el QUINTO GRUPO.` [2]

Extraer los bytes de los ficheros que contiene un directorio. Para este ejercicio utilice la salida del comando: **ls -l | grep** ^- como entrada de su expresión regular.

La solución pasa por implementar algo como: `4 grupos de un conjunto indefinido de caracteres distintos de 'espacio' seguidos de un conjunto indefinido de espacios; un grupo de números indefinido y un conjunto de caracteres libres hasta el final de la línea. Extraer el SEGUNDO GRUPO.` [3]

Ahora probaremos con un ejercicio incremental. Vamos a intentar parsear una dirección de correo tal y como la define el estándar (RFC822 (ftp://ftp.rfc-editor.org/in-notes/rfc822.txt)) vigente en Internet. Según ese estándar una dirección tiene las siguientes formas:

```
parte_local@dominio.completo
<parte_local@dominio.completo>
'nombre_real' <parte_local@dominio.completo>
"nombre_real" <parte_local@dominio.completo>
```

> **Nota:** Realmente el estándar define más formas, pero para no complicarlo, lo dejaremos ahí. Una cosa peculiar es que tanto la parte local como el dominio no pueden contener ciertos caracteres. (espacios, comas, signos mayor y menor que...)

Para añadirle una pizca de realismo, tomaremos la entrada como si fuera el contenido de una cabecera de un mensaje de correo electrónico (From:) de forma que nuestros casos a analizar siendo FORMA la forma de la dirección que estemos usando en cada momento:

```
From: FORMA
# ... y cualquier combinación de mayúsculas y minúsculas en la palabra From
```

En este caso iremos del caso más simple al más complicado; tomemos: `From: parte_local@dominio.completo`. Esta será la primera de las cuatro expresiones regulares que haremos.

Esta es facil: `Un conjunto indefinido de letras mayúsculas y minúsculas seguido de : y seguido de un número indefinido de espacios. Sustituir eso por NADA. Es decir, borrarlo.` [4]

Tenemos que darnos cuenta que necesitamos una expresión regular que sea capaz de extraer el email de cualquiera de las formas anteriores, así que debe ser lo suficientemente genérica.

Continuaremos con la siguiente, esta tambien es simple: `Un conjunto indefinido de letras mayúsculas y minúsculas seguido de : y seguido de un número indefinido de espacios. Un posible <. Un conjunto indefinido de caracteres distintos de: espacio, coma, y >. Un posible >. Extraer el SEGUNDO GRUPO.` [5]

Como vamos a empezar a tratar las comillas (" y ’) en las siguientes expresiones regulares para seguir resolviendo las distintas formas de la cabecera From:, es muy importante el entrecomillado que hagamos. Note el lector que para mayor comodidad y para que la concentración recaiga sobre las expresiones regulares, he entrecomillado las siguientes soluciones.

La tercera se complica un pelín: `Un conjunto indefinido de letras mayúsculas y minúsculas seguido de : y seguido de un número indefinido de espacios. Un conjunto formado por ’ cualquier número de caracteres y ’ que puede o no existir. Un conjunto formado por un número indefinido de espacios. Un posible <. Un conjunto indefinido de caracteres distintos de: espacio, coma, y >. Un posible >. Extraer el CUARTO GRUPO.` [6]

La última es relativamente sencilla si se consiguió la anterior; solo hay que añadir la posibilidad de que las comillas simples, también sean dobles.

Para terminar: `Un conjunto indefinido de letras mayúsculas y minúsculas seguido de : y seguido de un número indefinido de espacios. Un conjunto formado por ’ o " cualquier número de caracteres y ’ o " que puede o no existir. Un conjunto formado por un número indefinido de espacios. Un posible <. Un conjunto indefinido de caracteres distintos de: espacio, coma, y >. Un posible >. Extraer el CUARTO GRUPO.` [7]

Notas

```
s:.*\[\([0-9]*\)\]\:.*$:\1:g

s:\([a-zA-Z]\)\{3\}[ ]*\([0-9]\)\{1,2\} \([^ ]*\) \([^ ]*\) \([^[]*\).*:\5:g

s:^\([^ ]*[ ]*\)\{4\}\([0-9]*\).*$:\2:g

s:\([a-zA-Z]*\:[ ]*\)::g

s:\([a-zA-Z]*\:[ ]*\)<\{0,1\}\([^ ,>]*\)>\{0,1\}:\2:g

"s:\([a-zA-Z]*\:[ ]*\)\(’.*’\)\{0,\}\([ ]*\)<\{0,1\}\([^ >,]*\)>\{0,1\}:\4:g"

"s:\([a-zA-Z]*\:[ ]*\)\([\"’].*[’\"]\)\{0,\}\([ ]*\)<\{0,1\}\([^ >,]*\)>\{0,1\}:\4:g"
```

Capítulo 14. Scripts de inicio del sistema y ejecución programada de comandos.

Introducción

Ahora que conocemos suficientes comandos como para hacer varias cosas útiles en el sistema, hacer que algunos se ejecuten automáticamente es una necesidad inmediata.

Una de las formas posibles es que estos comandos se ejecuten siempre que arranquemos el sistema. La principal utilidad de todo esto son los demonios (la sección de nombre *Qué son*), principalmente comandos que ofrecen servicios (web, ftp, correo...). Lo ideal sería no tener que preocuparnos cada vez de iniciarlos, sino que siempre estén activos. Esto lo conseguiremos con los scripts de inicio.

Otra de las formas es hacer que se ejecuten periódicamente (una vez cada hora, una vez al día, ciertos días de la semana a la hora especificada...). El encargado de facilitarnos esta tarea será en este caso el demonio **crond**.

Scripts de inicio del sistema. Runlevels

La mayoría de las distribuciones de Linux se rigen por el *System V Init* para gestionar los scripts de arranque, así que es este el sistema que describimos aquí. A pesar de todo hay pequeñas diferencias de una distribución a otra, principalmente relativas a en qué directorio están determinados ficheros. En cualquier caso no son difíciles de encontrar. Otras distribuciones, como Gentoo o Slackware, tienen el método de inicio bastante distinto (Gentoo es SystemV modificado, y Slackware usa el BSD Init), así que en caso de necesitar más información en estos casos, lo más conveniente es buscar la documentación específica de la distribución.

Qué son los *runlevels*. Directorios

Los RUNLEVELS o también llamados NIVELES DE EJECUCIÓN son un concepto importante en los sistemas UNIX y por extensión en Linux. Hay varios runlevels disponibles, del 0 al 6 (puede haber alguno más, pero generalmente estos son los que se usan), y *cada uno de estos runlevels es una configuración de arranque distinta*. ¿Qué quiere decir esto? Entendámoslo bien. Pongamos por caso que tenemos una máquina que tiene dos usos, PC de escritorio y servidor de red. Estableceríamos entonces dos runlevels distintos. En uno de ellos configuraríamos para arrancar simplemente el servidor gráfico XWindow y un entorno de ventanas, y en el otro los distintos demonios servidores: servidor web, servidor DNS...

Cada uno de estos runlevels tendría un número distinto. Si durante una época determinada la máquina sólo va a funcionar como servidor, la configuraríamos para usar el runlevel correspondiente, que sólo cargaría al inicio los demonios de los servidores, y no cargaría el servidor gráfico de ventanas XWindow puesto que no lo necesita. Si durante otra época va a funcionar como PC de escritorio, la configuraríamos para usar el otro runlevel y nos evitaríamos tener que estar deteniendo los demonios que no necesitamos.

El runlevel 0 es para apagar la máquina, y el runlevel 6 es para reiniciarla. Es obvio que no debemos poner el número de runlevel por defecto a ninguno de estos dos; de hacerlo, el sistema no llegaría a arrancar. Otros runlevels típicos son el 3 y el 5 (generalmente significan arrancar el sistema en modo multiusuario; a veces pueden ser equivalentes, o el 5 arrancar el servidor gráfico X y el 3 no hacerlo

-- el equivalente en Debian suele ser el 2). El nivel 1 queda reservado para el modo monousuario que nos servirá para administrar la máquina; normalmente este modo no tendrá red y pedirá la password de root antes de permitir ejecutar ningún comando.

En `/etc/init.d` residen unos scripts, uno por cada servicio, que aceptan siempre un primer argumento, que puede ser "start" o "stop" (arrancar o detener el servicio). En cada runlevel podremos decidir cuáles de estos scripts se ejecutarán y cuáles no.

Para hacer esto, cada runlevel tiene un directorio asociado, `/etc/rcN.d` o bien `/etc/rc.d/rcN.d` dependiendo de la distribución de Linux usada, donde N es el número de runlevel correspondiente. Dentro de cada uno de ellos, hay una serie de enlaces simbólicos que apuntan hacia los scripts de los servicios en `/etc/init.d`. Para un runlevel determinado, sólo habrá enlaces para los servicios que se quieran ejecutar en ese runlevel.

Los nombres de esos enlaces son `ANNservicio`, donde A es una letra, y NN es un número. El objetivo de esta forma de nombrarlos es que en un runlevel determinado se ejecutan en orden alfabético. Así, para una misma letra, un número bajo indica que será ejecutado antes.

init, el primer proceso

Inmediatamente después de que el kernel se cargue en memoria, inicialice el hardware adecuadamente, y monte el sistema de ficheros, el comando **/sbin/init** da lugar al primer proceso del sistema, `init`, que es el padre de todos los procesos y el encargado de gestionar la configuración de los runlevels para iniciar los servicios adecuados.

init lee el fichero `/etc/inittab` (comentado a continuación). De él determina cuál es el runlevel por defecto (en el que iniciará), y algunas opciones.

Antes de iniciar el runlevel correspondiente, lo más común es que se ejecute un script (independientemente del número de runlevel). El nombre y la ubicación de este script se especifican también en `/etc/inittab`. Su misión es realizar tareas que siempre son necesarias; inicializar el teclado, los puertos serie, resolver las dependencias de los módulos, sincronizar el reloj del sistema con el reloj del hardware, definir el nombre del host... y cosas similares.

A continuación, **init** recurre a los enlaces simbólicos de `/etc/rcN.d` (o equivalente), ejecutando cada uno de ellos con un primer argumento, en este caso, "start".

Esto arranca todos los servicios correspondientes del runlevel (creando los nuevos procesos correspondientes). Seguidamente ejecuta varios terminales que esperan el login de los usuarios (las terminales a las que accedemos con Ctrl+Alt+F1...). Esto también está descrito en el fichero `/etc/inittab`.

El sistema ya ha arrancado y ahora es usable.

El fichero de configuración de init: `/etc/inittab`

En este fichero definiremos qué es lo que el sistema ejecutará en cada runlevel y todos esos extraños scripts de `/etc/init.d` empezarán a tener sentido.

El fichero tiene diversas entradas, del tipo:

```
identificador:runlevels:acción:orden_a_ejecutar
```

Donde:

- `identificador` es un nombre asociado a cada línea del fichero generalmente no superior a 4 caracteres.
- `runlevels` es una lista de runlevels (todos seguidos) para los cuales esta línea se tendrá en cuenta.
- `acción` indica de qué forma o bajo qué condiciones se ejecuta el comando del campo siguiente.
- `orden_a_ejecutar` es el comando que se ejecuta para esta entrada en los runlevels especificados.

De los campos anteriores, el campo `acción` merece especial interés, sobre todo porque algunos valores de este campo hacen que se ignore el campo `runlevels`. Las distintas opciones de este campo se pueden consultar en la página **man inittab**.

Lo mejor es ver algún ejemplo típico de un fichero `/etc/inittab` para ver cómo se combinan las diversas opciones. Este fichero suele tener comentarios que clarifican cómo están dispuestos los scripts de inicio por la distribución de Linux que estemos usando. En el ejemplo, los números indicados representan puntos que se comentan a continuación.

```
# Número de runlevel por defecto ❶

id:2:initdefault:

# Script que se ejecuta siempre en cada inicio,
# independientemente del runlevel ❷

si::sysinit:/etc/init.d/rcS # (Debian y similares)
si::sysinit:/etc/rc.d/rc.sysinit # (Redhat y similares)

# Comando para cada runlevel ❸
# En RedHat y sucedáneas es rc.d, Debian es init.d

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6


# Qué hacer en todos los runlevels si se presiona la combinación
# de teclas CTRL+ALT+DEL en una terminal ❹

ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now

# Lanzar las terminales para el login de los usuarios ❺

1:2345:respawn:/sbin/getty 38400 tty1
2:23:respawn:/sbin/getty 38400 tty2
3:23:respawn:/sbin/getty 38400 tty3
4:23:respawn:/sbin/getty 38400 tty4
5:23:respawn:/sbin/getty 38400 tty5
6:23:respawn:/sbin/getty 38400 tty6
```

Este podría ser un fichero de configuración de **init** a grandes rasgos. Algunas distribuciones añadirán más cosas, por ejemplo podrían poner un campo que ejecutase **xdm** (el programa de login gráfico) en el runlevel especificado, mientras que otras podrían tenerlo como un servicio aparte. Estudiemos el fichero por partes:

❶ En el ejemplo, el runlevel 2 es en el que iniciará el sistema por defecto. Puede editarse este fichero y cambiarse por otro.

❷ Existen una serie de operaciones, como ya vimos, que son necesarias en todos los runlevels, así que el script indicado en esta línea será ejecutado antes de cargar cualquier runlevel (esto lo causa "sysinit" al estar en el campo `acción`).

Esto está resuelto de formas distintas en distribuciones distintas, pero el script indicado tendrá comentarios y referencias que nos ayudarán a entender cómo funciona en nuestro sistema.

En RedHat y similares, es este fichero el que contiene los comandos a ejecutar directamente. En Debian, el script ejecuta todos los scripts que haya dentro del directorio `/etc/rcS.d` por orden alfabético, lo cual resulta ciertamente más ordenado.

❸ Para cada runlevel se ejecuta el comando **rc** (cuya localización puede variar según la distribución) con un argumento, el número de runlevel en cuestión. Así, este comando se encarga de ejecutar los scripts enlazados en el directorio `rcN.d`, donde N es el número de runlevel, pasándoles generalmente el argumento "start" para iniciar estos servicios.

La acción "wait" indica a **init** que debe esperar a que la ejecución de **rc** termine, esto es, a que se hayan iniciado los servicios.

❹ Esta línea asocia el evento de la pulsación de la combinación de teclas Ctrl+Alt+DEL con el comando especificado, en este caso, reiniciar la máquina mediante **shutdown**.

❺ Estas líneas inician las terminales en los runlevels indicados. El comando que lo hace puede ser **getty** o alguno similar. `tty1` a `tty6` son las terminales locales accesibles mediante Ctrl+Alt+F1 a F6. `tty7` suele estar reservado para utilizar el servidor gráfico X-Window. Estas terminales esperan a que les sea introducido un nombre de usuario y después llaman a **login** para autentificar al usuario y devolverle una shell válida en caso de éxito. Algunos comandos similares a **getty** permiten lanzar terminales en los puertos serie (incluso permitirían un inicio de sesión remoto a través de la línea telefónica mediante un módem).

La acción "respawn" indica que si el proceso de la terminal se detiene (por ejemplo, el usuario termina la sesión en una terminal), se debe volver a iniciar pidiendo un nombre de usuario de nuevo.

Cambio de runlevel

Con el sistema en marcha, es posible cambiar de runlevel en un momento determinado. Esto causará la detención de los procesos del runlevel actual e iniciará los del nuevo runlevel (si alguno de los procesos actuales es común a ambos runlevels no será parado y luego iniciado de nuevo). Si un proceso de los que deben detenerse no lo hace, será matado por **init**.

En un momento determinado puede interesar más detener un sólo proceso que efectuar un la sección de nombre *Cambio de runlevel*.

Para cambiar de runlevel se utilizará:

```
# telinit -t SEC NUM
```

En realidad, es equivalente **telinit** e **init** en este caso. Aunque tradicionalmente se viene usando **telinit** para cambiar de nivel de ejecución.

`SEC` es el número de segundos que esperará **init** a que un proceso se detenga antes de matarlo. Si no se especifica esta opción, el número de segundos por defecto es 5.

`NUM` es el número de runlevel o nivel de ejecución al que se quiere cambiar.

Un usuario no privilegiado no podrá cambiar de runlevel (por otra parte es obvio, si no le pertenecen esos procesos no tendrá ningún derecho sobre ellos).

Un claro ejemplo de cambio de runlevel es cuando utilizamos alguno de los comandos que nos permiten apagar la máquina. En definitiva lo que hacen es un cambio al runlevel 6.

Re-lectura del fichero de configuración

Si en algún instante queremos que **init** vuelva a leer el fichero de configuración, tendremos que indicárselo mediante:

```
# telinit q
```

Modo monousuario

Como cabía esperar, **init** también está relacionado con el establecimiento de la jerarquía de permisos al inicio del sistema. Normalmente se iniciará la máquina en modo *multiusuario* con todas las ventajas y características que ya hemos visto. Sin embargo, existe la posibilidad de iniciar la máquina Linux en modo monousuario.

Qué pasará o qué servicios se inicializarán en modo monousuario depende de cada distribución de Linux. En algunas incluso es posible que se muestre un prompt con permisos de root sin ni si quiera pedir un login.

En principio, poner un sistema en modo monousuario carece de utilidad práctica; aunque no está demás saber que esta posibilidad existe, y también, que no es segura.

Más información

Cada sistema puede tener una configuración ligeramente distinta en este aspecto. Nuestro papel, si administramos una máquina, es conocer cómo está configurado el arranque para poder modificarlo según nuestras necesidades.

Una buena fuente de información al respecto son los comentarios que hay en los ficheros de `/etc/init.d` (o equivalente) y `/etc/initab`, que en la mayoría de las ocasiones explican cómo se ha personalizado el arranque por los creadores de la distribución.

Las páginas man de **init** e **inittab** son también referencia obligada si se desea profundizar más en estos aspectos.

Servicios, demonios

Servicios y demonios guardan una estrecha relación con los niveles de ejecución. Descubrámoslo.

Qué son

El nombre de "servicio" es un poco ambiguo pero generalmente se refiere a un demonio del sistema. El glosario define el término *Demonio*.

Cada uno de ellos tiene un archivo asociado en `/etc/init.d` que como ya se vió es un script bash que acepta un argumento (podría ser, al menos, "start" o "stop", aunque pueden ser otros complementarios para proveer al usuario de opciones adicionales, por ejemplo "restart", "status"...).

Este script asociado ejecutará el comando que inicializa el demonio con las opciones adecuadas si se le pasa "start" como argumento.

Si observas cualquier script de un servicio que haya instalado tu distribución podrás entender el esquema general que siguen todos ellos.

Añadir y quitar servicios a un runlevel

Existen dos posibilidades de hacer esto. Una es la vía manual (haremos enlaces simbólicos), y la otra usando las vías que nos facilite nuestra distribución.

Veremos varios casos:

- Poner o quitar de un runlevel programas instalados por nuestra distribución. Estos ya tendrán un fichero generalmente `/etc/init.d/servicio` donde *servicio* será generalmente (y esto es arbitrario, no es imprescindible) el nombre asociado al demonio o servicio que inicie.
- Poner o quitar de un runlevel programas que hayamos compilado desde las fuentes, creados por nosotros o similar. En este caso nosotros crearemos el script en `/etc/init.d` y los enlaces simbólicos.

Fichero correspondinete en `/etc/init.d`

La estructura de estos ficheros es, a grandes rasgos:

```
#!/bin/bash

# Comentarios de los autores

# Definición de variables de shell que dicen dónde esta el
# ejecutable del demonio, el fichero de configuración, o detalles
# de este estilo

# Comprobaciones de si existe el fichero ejecutable del programa,
# si existen los ficheros de configuración, si determinados permisos
# tienen los ficheros adecuados, etc.

case "$1" in

 start)
   echo "Iniciando nombre_servicio... "
   orden_de_shell # Comando que inicializa el servicio
   ;;

 stop)
   echo "Parando nombre_servicio... "
   orden_de_shell # Comando que detiene el servicio
```

```
    ;;

 restart)
    echo "Reiniciando nombre_servicio... "
    orden_de_shell # Comando que reinicia el servicio
    ;;

 # Otras opciones posibles aceptadas (diferentes para cada servicio) como:
 # status: Daría información sobre el estado del servicio
 # reload: Enviaría una señal al proceso para releer la configuración

 *)
    echo "Uso: $0 {start|stop|restart|(otras opciones)}
    ;;

esac

exit 0
```

orden_de_shell es el comando correspondiente. Este comando puede ser de dos tipos:

- Directamente referirse al programa e iniciarlo con las opciones adecuadas.
- Iniciarlo indirectamente llamando a un programa o función de bash (según distribución) que además de ejecutarlo guarde en un fichero su ID de proceso para posteriormente detenerlo con más facilidad.

Por ejemplo, para el segundo caso, en Debian está el programa **start-stop-daemon** y en RedHat (y derivados) está definida una función Bash en `/etc/init.d/functions` llamada **daemon**. En el primer caso puedes leer su página man y en el segundo puedes ver directamente su definición en el fichero indicado para ver cómo funciona, o fijarte en scripts de inicio ya creados.

Cuando no se usa la segunda opción, para terminar el proceso se usa el comando **killall** para enviarle una señal indicando sólo el nombre del proceso.

Servicios que instala nuestra distribución

Estos ya tienen su fichero en `/etc/init.d`. Se iniciarán en los runlevels en los que tengan enlace simbólico en el directorio correspondiente a su número de runlevel. Por ejemplo, añadamos el servidor web Apache para que se inicie en el runlevel 3:

```
# ln -s /etc/init.d/httpd /etc/rc3.d/S80httpd
```

"S80" es arbitrario (recuerda que se ejecutan por orden), pero debe ser lo suficientemente "alto" para que los servicios de red ya estén iniciados.

Para hacer que un servicio no se ejecute en el runlevel 3, borraremos su enlace en `/etc/rc3.d`

Servicios nuevos

Con esto nos referimos a los demonios que, por ejemplo, instalemos compilando el código fuente. Este procedimiento de instalación, al contrario que los paquetes de las distribuciones, no crea un fichero adecuado en `/etc/init.d`.

Es nuestro turno: miraremos la documentación del paquete para ver con qué comando se inicia y con cuál se detiene, y crearemos un fichero en `/etc/init.d` con la misma estructura que el que hemos visto arriba. Le daremos permisos de ejecución y lo llamaremos con un nombre que nos permita identificarlo posteriormente con el demonio que iniciará.

A continuación se procede exactamente igual que en el apartado anterior.

Ver los servicios que se iniciarán en un runlevel

Es tan sencillo como listar los enlaces que hay en `/etc/rcN.d` donde N es el número de runlevel.

Arrancar y parar servicios por separado

Con el sistema iniciado en un runlevel determinado, puede surgirnos la necesidad de activar o desactivar un sólo servicio. Para esto es obvio que no merece la pena cambiar de runlevel.

Incluso podemos, desde la consola como root, ejecutar un **kill** para terminar un servicio refiriéndonos a su proceso; o poner el comando del demonio que queremos iniciar y hacer que se ejecute en segundo plano.

No obstante, aunque lo anterior es perfectamente válido, no es necesario que recordemos todas las opciones necesarias de un demonio, o su ID de proceso si los scripts de `/etc/init.d` tienen ya esta información.

Así, podríamos iniciar, por ejemplo, el servidor de bases de datos MySQL con:

```
# /etc/init.d/mysqld start
```

En RedHat y sucedáneos, este comando es equivalente a **service mysqld start**.

Exactamente de la misma forma se puede detener un servicio:

```
# /etc/init.d/httpd stop
```

dentendría el servicio httpd suponiendo que este ya estuviese iniciado.

Crond, ejecución programada o periódica de comandos

¿Qué es?

Cron es una utilidad, presente en la mayoría de sistemas Linux y Unix, que permite ejecutar comandos o procesos con una frecuencia determinada.

Esto se puede aplicar, por ejemplo, a hacer backups de una carpeta del sistema todos los días a las 12 de la noche, comprobar el tamaño de un directorio o el espacio libre en disco de los usuarios del sistema y avisar cuando está a punto de agotarse, ejecutar un revisor de correo como fetchmail y mil usos más: todo queda limitado por tu imaginación ;)

¿Y esto cómo funciona?

Cron se compone principalmente de 2 elementos: **cron**, el *Demonio*, y el archivo de configuración `/etc/crontab`.

Si pretendemos añadir, quitar o modificar las tareas que ejecutará **cron**, se puede hacer uso del siguiente comando:

```
$ crontab -e
```

con el cual se abrirá un editor y podremos cambiar nuestas tareas en el cron. La diferencia entre editar el archivo `/etc/crontab` directamente y utilizar el comando **crontab -e** radica en que con este último no hacen falta privilegios de root la sección de nombre *Usuarios* en Capítulo 5 ya que estaremos editando un archivo de configuración específico para nuestro usuario. Una vez que se ha acabado de editar el archivo, al guardarlo, **crontab** se encarga de integrar todo en cron. ¿Cómo editarlo? Pues bien, hay una forma definida de hacerlo en la que cada línea tiene este orden:

```
minuto hora dia_del_mes mes dia_de_la_semana comando
```

Y estos son los posibles valores para cada elemento:

Tabla 14-1. Valores para `/etc/crontab`

Minuto	Entre 0 y 59
Hora	Entre 0 y 23
Dia del mes	Entre 1 y 31
Mes	Entre 1 y 12
Dia de la semana	Entre 0 y 6. El 0 corresponde al Domingo, el 1 al Lunes y así hasta el 6 que corresponde al Sábado
Comando	Se trata del comando o comandos a ejecutar.

Si no quieres especificar un valor concreto para alguna de las variables, se debe colocar el signo * el cual significa algo así como "todos" los valores. Si lo que quieres es especificar un rango, has de usar el signo - de este modo: 0-5 ó 6-9 o similar. Si por el contrario deseas indicar varios valores por separado puedes hacer uso de las comas: 4,7,9,15 . Y si quieres excluir algún valor se usa / : 0-5/4 de modo que se ejecutará de 0 a 5 excepto el 4 (depende de si te refieres a minutos, horas, dias de la semana...). El signo / también puede ser usado para determinar intervalos, por ejemplo cada 15 minutos: */15.

Otro aspecto a tener en cuenta es que si el comando no está en el *PATH* deberá ser especificado con su ruta completa. Veamos algunos ejemplos:

```
0,5,10,15,20,25,30,35,40,45,50,55 * * * * fetchmail
```

```
*/5 * * * * fetchmail
```

En estos ejemplos, se ejecutará el programa fetchmail todos los días del mes, todos los días de la semana, todos los meses, cada hora, cada 5 minutos.

```
0 4 * * * fetchmail
```

El comando se ejecutará todos los días a las 4:00 de la mañana.

```
0 12 * * 3 shutdown -h now
```

Con esta línea, el computador se apagará todos los Miércoles a las 12 del mediodía.

Capítulo 15. Shell scripting II.

Introducción

.

Glosario de términos

Algunas de las definiciones están sacadas del glosario de términos de E.C.O.L (http://escomposlinux.org)

A

Adduser

El comando **adduser** lo utiliza root, o alguien que tenga autoridad para ello, con el fin de crear un nuevo usuario. Al comando **adduser** le sigue el nombre de cuenta que se va a crear. Cabe destacar que en algunas distribuciones este comando a cambiado su nombre por **useradd**.La sintaxis de este comando vendría a ser por ejemplo:

adduser nombre_usuario

Alias

El comando alias se usa para crear alias o nombres alternativos para comandos. Generalmente, estos alias son abreviaturas del verdadero comando. Dichos alias se situan en el `.bashrc` de cada usuario, por ejemplo alias **rmd="rm -Rf"** haría que al teclear el comando **rmd** nos borrara lo que indiquemos en modo recursivo.

Apropos

El comando **apropos** significa literalmente apropiado o relativo (a otros). Cuando le sigue un parámetro, buscará en las páginas man entradas que incluyan el parámetro. Básicamente, con esto se realiza una búsqueda de palabras clave en todas las páginas man. Es el equivalente del comando **man -k parametro**.

B

Bash

(Bourne Again SHell) Intérprete de comandos. Es el shell por defecto en la mayoría de las distribuciones de GNU/Linux de hoy en día. Se encarga de interpretar los comandos del usuario para llamar a los programas que están en los directorios de la variable `$PATH` o en los alias, a parte de ejecutar sus propios comandos: asignación de variables, bucles, etc.

Buffer

Según el diccionario, "memoria intermedia". Es una memoria que ciertos programas usan para diversas funciones, normalmente para guardar datos en memoria para su uso en un plazo corto de tiempo.

BIOS

Sigla de Basic Input/Output System (sistema de entrada/salida básico). Se utiliza para realizar todas las funciones necesarias para colocar en estado inicial el hardware del sistema cuando se conecta a la alimentación de energía. El BIOS controla el proceso de arranque, proporciona rutinas de entrada/salida de bajo nivel (de aquí su nombre) y (usualmente) permite que el usuario modifique los detalles de la configuración del hardware del sistema.

C

Cat

cat le indica al sistema que "concatene" el contenido de un archivo con la salida estándar, normalmente la pantalla. Si ese archivo es binario, el comando **cat** se puede liar y la salida puede no ser muy sugerente. Generalmente, este proceso también es muy ruidoso. Lo que en realidad ocurre es que el comando **cat** está desplazándose por los caracteres del archivo, y el terminal está haciendo todo lo que puede para interpretar y mostrar los datos del archivo. Tendría básicamente un formato del tipo **cat fichero** .

Código fuente

El formato entendible por las personas de las instrucciones que conforman un programa. También se le conoce como «fuentes» o «source code».

Comando

Medio por el cual se le ordena una acción determinada al sistema operativo a través de un intérprete de comandos, tal y como puede ser Bash.

Compilar

Proceso por el cual se "traduce" un programa escrito en un lenguaje de programación a lo que realmente entiende el ordenador.

D

Demonio

Aparte del significado que todos conocemos, en Unix/Linux se conoce como un programa que permanece en segundo plano ejecutandose continuamente para dar algun tipo de servicio. Ejemplos de demonio, son los servidores de correo, impresora, sistemas de conexion con redes, etc.

Dependencias

Cuando se refiere a paquetes, las dependencias son requerimientos que existen entre paquetes. Por ejemplo, el paquete "foo" puede requerir ficheros que son instalados por el paquete "bar". En este ejemplo, "bar" debe estar instalado, pues sino "foo" tendrá dependencias sin resolver. Normalmente, RPM no permitirá que se instalen paquetes con dependencias sin resolver.

Distribución

Un sistema operativo (en general Linux), que se ha empaquetado para facilitar su instalación.

Distro

Ver *Distribución*.

DNS

Domain Name Server. Servidor de nombres de dominio. Servicio de red que nos facilita la busqueda de ordenadores por su nombre de dominio. Se encarga tanto de traducir nombres a direcciones IP como del paso contario.

E

Entrada/Salida estándar

Por defecto la entrada de datos estándar se establece en el teclado y la salida de datos estándar en la pantalla del monitor, esto lo podemos variar a través de tuberías. Por ejemplo, podemos hacer que la entrada sea el ratón y la salida la impresora.

Expresión Regular

Conjunto de caracteres que forman una plantilla para buscar y reemplazar cadenas de texto dentro de textos más largos.

F

FSF

Free Software Foundation. Fundación que pretende el desarrollo de un sistema operativo libre tipo UNIX. Fundada por Richard Stallman, empezó creando las herramientas necesarias para su

proposito, de modo que no tuviera que depender de ninguna compañia comercial. Después vino la creación del núcleo, que todavía se encuentra en desarrollo.

G

GNU

Gnu is Not Unix. Proyecto de la FSF para crear un sistema UNIX libre.

GNU/Linux

Sistema operativo compuesto de las herramientas GNU de la FSF y el núcleo desarrollado por Linus Torvalds y sus colaboradores.

GPL

General Public License. Una de las mejores aportaciones de la FSF. Es una licencia que protege la creación y distribución de software libre.

H

Hardware

El hardware es el soporte físico de una computadora, se compone de diferentes dispositivos de hardware que pueden estar dentro o fuera de la caja de la computadora. Dentro del hardware entran la placa base, la cpu, la memoria, el teclado, el ratón...

Host

Un host es una computadora que se encuentra dentro de una red, y que ofrece algún tipo de servicio o recurso al resto.

I

Intérprete de comandos

Ver *Shell*.

IP

Las direcciones IP (Internet Protocol) son el método mediante el cual se identifican los ordenadores individuales (o, en una interpretación más estricta, las interfaces de red de dichos ordenadores) dentro de una red TCP/IP. Todas las direcciones IP (versión 4, la más utilizada actualmente) consisten en cuatro números separados por puntos, donde cada número está entre 0 y 255.

A veces, la abreviatura IP puede significar "Intelectual Property", o lo que es lo mismo, Propiedad Intelectual.

ISP

Siglas de Internet Service Provider. Empresa u organización que ofrece acceso a Internet a usuarios finales y corporativos.

K

Kernel

Parte principal de un sistema operativo, encargado del manejo de los dispositivos, la gestión de la memoria, del acceso a disco y en general de casi todas las operaciones del sistema que permanecen invisibles para nosotros.

L

Librerías

Se refiere al conjunto de rutinas que realizan las operaciones usualmente requeridas por los programas. Las librerías pueden ser compartidas, lo que quiere decir que las rutinas de la libreria residen en un fichero distinto de los programas que las utilizan. Las rutinas de libreria pueden «enlazarse estáticamente» al programa, en cuyo caso se agregan físicamente las copias de las rutinas que el programa necesita. Estos binarios enlazados estáticamente no requieren de la existencia de ningún fichero de biblioteca para poder funcionar. Los programas enlazados con bibliotecas compartidas no funcionarán a menos que se instalen las librerias necesarias.

Login

Programa encargado de la validación de un usuario a la entrada al sistema. Primero pide el nombre del usuario y despues comprueba que el password sea el asignado a este.

Logout

logout se utiliza para salir de un sistema como usuario en curso. Si es el único usuario que está registrado, se desconectará del sistema.

M

Módulos

Un módulo es un conjunto de rutinas que realizan funciones a nivel de sistema, y que pueden cargarse y descargarse dinámicamente desde el núcleo cuando sea requerido. Los módulos con frecuencia contienen controladores de dispositivos, y están fuertemente ligados a la versión del núcleo; la mayoría de los módulos construidos con una versión dada de núcleo, no se cargarán de manera apropiada en un sistema que corra un núcleo con versión distinta.

Multitarea

Capacidad de un sistema para el trabajo con varias aplicaciones al mismo tiempo.

Multiusuario

Capacidad de algunos sistemas para ofrecer sus recursos a diversos usuarios conectados a traves de terminales.

N

Núcleo

Ver *Kernel*.

P

Paquete

Fichero que contiene software; está escrito en un cierto formato que permite la fácil instalación y borrado del software.

Partición

El segmento del espacio de almacenamiento de una unidad de disco que puede accederse como si fuese un disco entero.

PATH

Variable del entorno, cuyo valor contiene los directorios donde el sistema buscará cuando intente encontrar un comando o aplicación. Viene definida en los ficheros `.bashrc` o `.bash_profile` de nuestro directorio home.

PPP

Point to Point Protocol. Protocolo de transmisión de datos, utilizado en la mayoria de las conexiones a internet domésticas.

Proceso

Programa en ejecucción dentro de un sistema informático.

Pipe

Ver *Tubería*.

Password

Palabra clave personal, que nos permite el acceso al sistema una vez autentificada con la que posee el sistema en el fichero `passwd`.

POSIX

POSIX es el acrónimo de Portable operating system interface, una familia de estándares de llamadas al sistema definidos por el IEEE, intenta estandarizar las interfaces de los sistemas operativos para que las aplicaciones corran en distintas plataformas. Estos estándares surgieron de un proyecto de estandarización de APIs y describen un conjunto de interfaces de aplicación aplicables a una gran variedad de implementaciones de sistemas operativos. El término POSIX fue sugerido por Richard Stallman en respuesta a la demanda de la IEEE que requería un nombre facilmente memorizable.

Prompt

El prompt es lo siguiente que vemos al entrar al sistema, una linea desde donde el sistema nos indica que esta listo para recibir ordenes, que puede ser tan sencilla como: $autoconf o algo mas compleja como: `amphora:1505200:home/israel:$` En la mayoría de las shells (incluida bash), es personalizable.

R

root

Persona o personas encargadas de la administración del sistema Tiene TODO el privilegio para hacer y deshacer, por lo que su uso para tareas que no sean absolutamente necesarias es muy peligroso.

Recursos del sistema

Engloba a la memoria RAM, el procesador y otros dispositivos de hardware como el disco duro. De los recursos del sistema depende el desempeño del ordenador.

S

Sistema Operativo (S.O.)

Un Sistema Operativo (S.O.) es un programa (o conjunto de programas) de control que tiene por fin facilitar el uso de la computadora y conseguir que ésta se utilice eficientemente. Así, también reúne un conjunto de funciones o programas que hacen la vida del programador más fácil, y sirve como plataforma en la que basarse, para no tener que reinventar la rueda con cada programa que se cree.

Shell

Traducido del inglés concha o caparazón. El shell es el intérprete de comandos que se establece entre nosotros y el kernel. Hay muchos tipos de shell cada uno con sus propias características, sin embargo el estándar en GNU/Linux es el shell bash ya que es el que forma parte del proyecto GNU.

Script

Un script es un fichero de texto que contiene un conjunto de comandos/instrucciones escritos en un lenguaje interpretado cualquiera que sea, como puede ser Bash. La peculiaridad de los scripts es que son programas que no están compilados, sino que contienen el código fuente del programa, que es interpretado y pasado a un lenguaje que la computadora pueda comprender cada vez que se mandan ejecutar.

Shell scripting

El shell scripting es el arte de programar script en una shell.

Shell scripts o scripts de shell

Programa escrito para ser interpretado por una shell de un SO, especialmente Unix.

Swap

Espacio de disco duro que utiliza el kernel en caso de necesitar mas memoria de la que tengamos instalada en nuestro ordenador.

T

Tubería

Las tuberías son como conexiones entre procesos. La salida de un proceso la encadenamos con la entrada de otro, con lo que podemos procesar unos datos en una sola linea de comando.

Terminal

Una terminal es un teclado y una pantalla conectados por cable u otro medio a un sistema UNIX/Linux, haciendo uso de los recursos del sistema conectado.

U

UNIX

Sistema Operativo creado por AT&T a mediados de los 70.

W

Wildcards

Las wildcards son unos caracteres especiales usados comúnmente dentro de búsquedas de ficheros por su nombre con comodines. Por ejemplo, si buscamos por un archivo nombrado como ’r?te’ en un Sistema Operativo tipo Unix, nos puede encontrar archivos de nombre ’rute’, ’rote’,’rzte’, etc. El asterisco en Unix se utiliza para substituir "cualquier cosa", por ejemplo ’archivo_*_inet’ puede encontrarnos cualquier archivo que empiece por ’archivo_’ y termine por ’_inet’ (como por ejemplo ’archivo_de_inet’, ’archivo_sdsfhy_inet’ o ’archivo__inet’)

Apéndice A. ZonaSiete.ORG Editors Team

Aquí están listados todos los editores que han ayudado, por poco que sea, a escribir este manual. Algunos siguen escribiendo, otros por razones diversas se han tomado un tiempo de descanso; quizá indefinido. De todas formas queremos agradecerles a *todos* su trabajo ya que sin ellos este trabajo no habría sido posible.

ZonaSiete.ORG Editors Team <zonasiete@ferdyx.org>

Editores en activo

Los siguientes editores están en activo:

- Ricardo Cervera Navarro <ricardo@zonasiete.org>
- Fernando J. Pereda Garcimartín <ferdy@ferdyx.org>
- Eduardo González López de Murillas <freenix@coaxion.org>
- Noé Ávila López <jaguar@forodelinux.org>
- Antonio Hernández González <toyohg@terra.es>

Editores retirados

Muchas gracias a todos, teneis la puerta abierta si decidís volver:

- Aitor Moreno Martínez <zesder@supercable.es>
- Ignacio Hernán Jerez <herje@zonalinux.org>
- Eduardo Robles Elvira <edulix@tumundoweb.com>
- Iván Calle Cabrero <ivy@ferdyx.org>

Colaboradores

Personas que nos han echado una mano en momentos puntuales:

- Sergio Pereda Garcimartín <sergio@ferdyx.org>
- Alejandro Cadavid López <acadavid@gmail.com>

Apéndice B. GNU Free Documentation License

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invari-

ant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and

disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.