Fundamentos del Software

Prácticas

Módulo I

Órdenes UNIX y Shell Bash

11-jul-2018

Grado en Ingeniería Informática y
Doble Grado en Ingeniería Informática y Matemáticas

Departamento de Lenguajes y Sistemas Informáticos
E.T.S. Ingenierías Informática y de Telecomunicación
Universidad de Granada

Contenido

Práctica 0: Introducción a un entorno gráfico de UNIX

0.1 Introducción

Esta parte se dedica a la instalación sencilla de un sistema operativo basado en Linux y al manejo de la interfaz de usuario para sistemas Unix/Linux

0.2 Objetivos principales

- Instalación de un sistema operativo Unix/Linux.
- Saber entrar y salir en los ordenadores del aula.
- Saber utilizar una sesión como usuario en un sistema Unix/Linux.
- Saber ejecutar programas y localizarlos en la interfaz.
- Conocer el manejo de rutas y los conceptos relacionados con este.

En las aulas se dispone de acceso identificado, mediante usuario (user) y contraseña de acceso (password).

0.3 Introducción y políticas de seguridad

Es importante comprender que un usuario y contraseña asocian a un usuario real con uno virtual, identificándolo en el sistema. Para ello es recomendable:

- Evitar contraseñas que tengan algún significado como nuestra fecha de nacimiento, nuestro teléfono, el nombre de nuestros familiares, etc.
- Evitar contraseñas que sean palabras o nombres propios en cualquier idioma, ya que hay sofisticados programas de ataques por diccionario que reúnen todos estos términos y los van comprobando sistemáticamente.
- No escribir nuestras contraseñas en un papel y menos dejarlo en un lugar accesible como al lado del teclado.
- Tampoco deberíamos enviar nuestras contraseñas por correo, messenger, etc.
- No debemos emplear la misma contraseña para todo.
- Para evitar el "phishing" nunca debemos responder correos donde nos soliciten nuestras contraseñas o datos y, menos aún, del banco.
- Trataremos que nuestras contraseñas sean de por lo menos 8 caracteres o más.
- La contraseña debe tener letras en mayúsculas, minúsculas y números. Si el sistema lo admitiera también sería recomendable añadirle algún carácter no alfanumérico como un corchete, un asterisco etc. Esto es debido a que existen programas de ataques por fuerza bruta que van probando todas las combinaciones posibles de manera que cuanto más larga sea nuestra contraseña y más variedad de elementos contenga el número de combinaciones crece exponencialmente.
- Cambiar periódicamente la Contraseña.

0.4 Instalación sencilla de un sistema operativo Linux

Dado que se interaccionará con un sistema operativo Unix/Linux a lo largo del periodo de aprendizaje de ésta y de otras asignaturas a lo largo de la titulación de grado, se considera de especial importancia ilustrar de qué forma se puede disponer de un ordenador con un sistema operativo Linux instalado. En las aulas de la ETSIIT no se podrán realizar instalaciones de ciertos programas pero sí en los ordenadores de sobremesa y/o portátiles de adquisición propia. En ese sentido, dado que muchos disponen de ese equipamiento en sus casas y así poder desempeñar parte del trabajo a realizar, mostraremos a continuación una forma de instalar un sistema operativo linux en esos

ordenadores que, en la amplia mayoría de los casos, dispone de un sistema operativo basado en windows (xp, vista, 7, etc.).

Si ya se dispone de un ordenador con un sistema operativo windows instalado, existen varias formas de instalar un sistema operativo linux disponible también en el mismo ordenador sin pérdida alguna de funcionalidades. Las opciones serían:

a) Particionar el disco del ordenador y disponer de un espacio para windows y otro para linux. Aquí sólo podríamos arrancar el ordenador eligiendo uno de los sistemas operativos instalados, es decir, no sería posible ejecutarlos de forma conjunta, con lo cual, pasar de un sistema a otro requeriría el reinicio del ordenador.

b) A partir de un sistema operativo instalado (windows, OS X o Linux) instalar un software de máquina virtual para montar cualquier otro sistema operativo. En este caso, se podría disponer de varios sistemas operativos ejecutados de forma conjunta dado que una máquina virtual no es más que otro proceso que se ejecutaría en nuestro sistema.

De las dos opciones anteriormente mostradas, se mostrará en este guión la opción (b) y, partiendo de un ordenador con windows instalado, proceder a instalar una máquina virtual y luego un sistema operativo linux.

0.4.1 Instalación de una máquina virtual (VirtualBox)

Según reza la wikipedia, VirtualBox es un software de virtualización para arquitecturas, creado originalmente por la empresa alemana innotek GmbH. Actualmente es desarrollado por Oracle Corporation como parte de su familia de productos de virtualización. Por medio de esta aplicación es posible instalar sistemas operativos adicionales, conocidos como «sistemas invitados», dentro de otro sistema operativo «anfitrión», cada uno con su propio ambiente virtual. ¿Qué se entiende por virtualización? bueno en informática virtualización es la posibilidad de estar ejecutando más de un sistema operativo simultáneamente. Esto quiere decir que podemos estar usando algún sistema operativo base como Linux, Windows o OS X y, sobre él, ejecutar un software de virtualización (VirtualBox). Dicho software utilizará parte de los recursos del sistema anfitrión para proporcionarselos al sistema operativo que se ejecute sobre dicha máquina virtual. Los recursos que se "piden prestados" son:

- Parte del *disco duro*, que constituirá un disco virtual sobre el que instalará el sistema (o sistemas) operativos que deseamos que puedan arrancar y funcionar sobre la máquina virtual.
- Parte de la *memoria principal* (RAM), para poder ejecutar el sistema operativo invitado y, por supuesto, las aplicaciones que se ejecuten sobre éste.
- Parte de la memoria de la *Tarjeta gráfica*, para poder utilizar los recursos de salida gráfica del hardware real.

Entre los sistemas operativos soportados (en modo anfitrión) se encuentran GNU/Linux, OS X, OS/2 Warp , Microsoft Windows, y Solaris/OpenSolaris, y dentro de ellos es posible virtualizar los sistemas operativos FreeBSD, GNU/Linux, OpenBSD, OS/2 Warp, Windows, Solaris, MS-DOS y muchos otros.

Para instalar VirtualBox en nuestro ordenador con windows sólo es necesario descargar el programa en cuestión del siguiente enlace: `https://www.virtualbox.org/wiki/Downloads`

Posteriormente, unos sencillos pasos proporcionarán la instalación de VirtualBox y la consiguiente posibilidad de instalar nuevas máquinas virtuales con sus correspondientes sistemas operativos. La primera ventana que nos aparece es la de bienvenida (ver figura 1.2). Si la instalación se trata, como es el caso, de una por defecto, únicamente se han de pulsar las teclas de Siguiente (Next) en varias pantallas que irán apareciendo hasta la aparición de una que nos advertirá de que tendrá que detener temporalmente la actividad de la red (ver figura 1.3). Pulsamos "Yes" y el proceso de instalación continuará sin problema alguno hasta completarse en pocos minutos.

Una vez concluida la instalación se inicia la aplicación donde aparecerá la siguiente pantalla de inicio (ver figura 1.4).

Figura 0.3. Pantalla de bienvenida del proceso de instalación de VirtualBox.

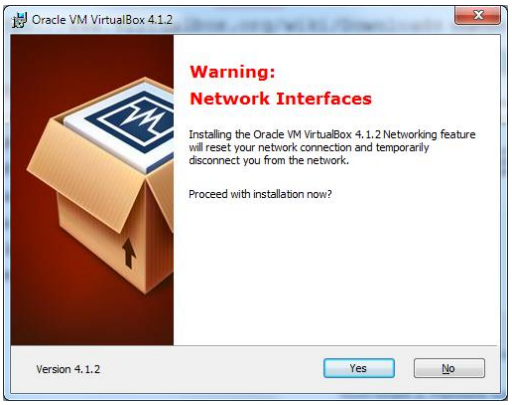


Figura 0.2. Advertencia durante el proceso de instalación de VirtualBox.

0.4.2 Instalación de Ubuntu Linux en VirtualBox

Una vez instalado VirtualBox, procederemos a la creación de la máquina virtual e instalación de Linux como sistema operativo. Al tratarse de una máquina virtual, hemos de definirle los parámetros más característicos de un ordenador de sobremesa o portátil, es decir, memoria RAM, memoria de video, tamaño del disco, etc. Todos esos detalles se ilustran a continuación.

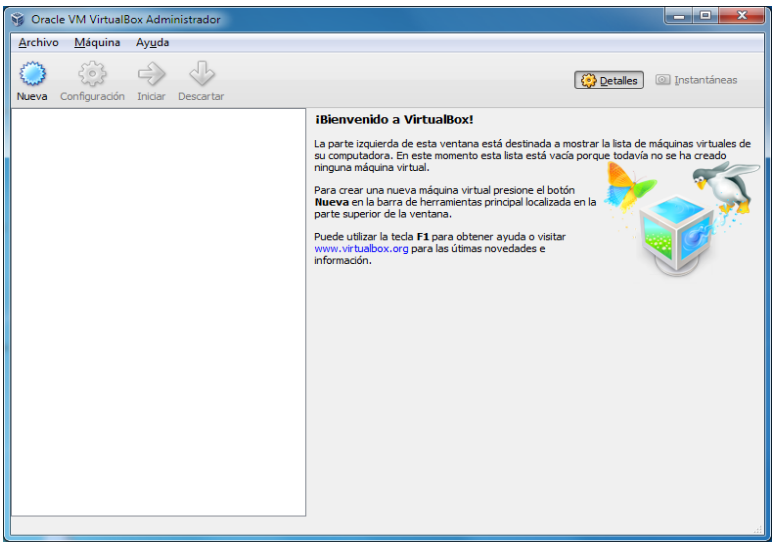
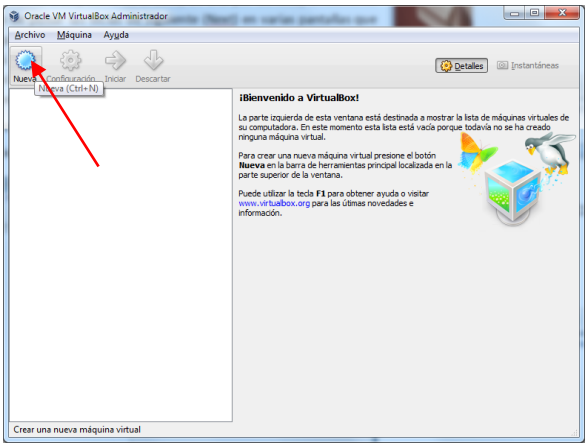


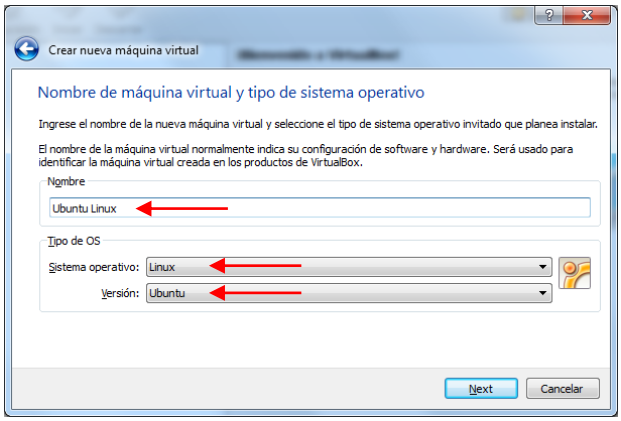
Figura 0.4. Ventana de inicio tras la primera ejecución de VirtualBox.

0.4.2.1 Determinación de la máquina virtual

En primer lugar hemos de pulsar la opción "Nueva" tal y como aparece en la siguiente figura.

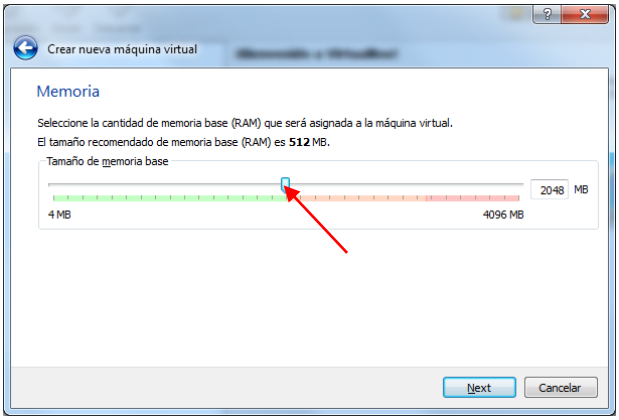


Tal y como se ha dicho al comienzo de esta sección, ahora habrá que definir los parámetros esenciales de esta máquina virtual. En la siguiente figura se indica que se va a crear una denominada Ubuntu Linux y, automáticamente, VirtualBox refleja que la máquina tendrá un sistema operativo basado en Linux con una distribución de ubuntu.



0.4.2.2 Determinación de la memoria virtual

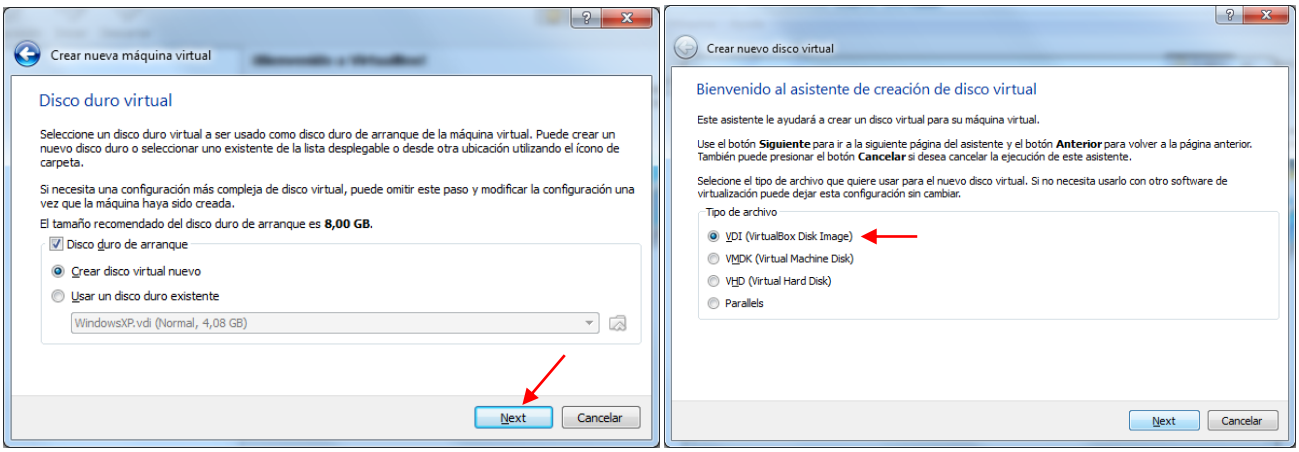
En primer lugar hay que determinar, del total de la memoria principal que tiene la máquina real, qué cantidad de memoria se reservará para la ejecución de la máquina virtual.



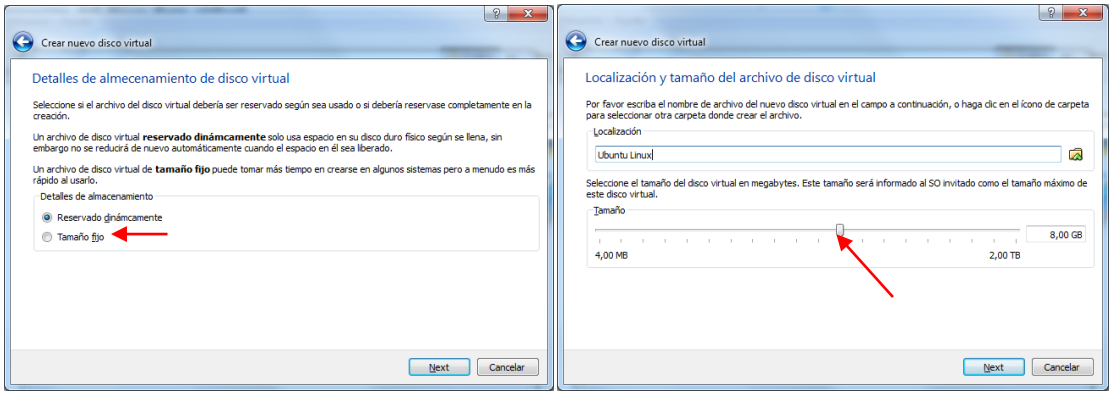
Aunque no existe una cantidad concreta, se aconseja usar, al menos, la mitad de la memoria principal que disponga nuestro ordenador real. En el ejemplo, de los 4GB de que dispone, se destinarán 2GB a la máquina virtual de Ubuntu.

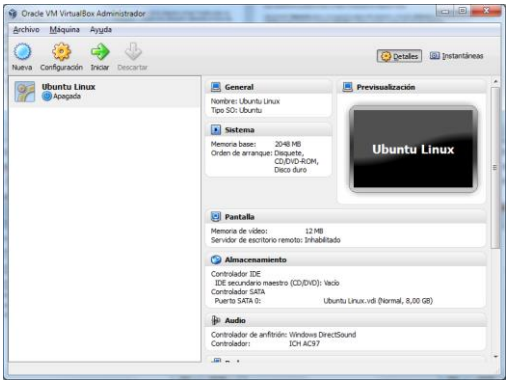
0.4.2.3 Determinación del disco duro virtual

Lo siguiente es definir el disco donde se alojará el sistema operativo a instalar y los datos del usuario. Al tratarse de una máquina virtual nueva, se ha de crear un nuevo disco virtual. Por lo tanto, pulsaremos "Next".



El disco virtual, en realidad, será un archivo dentro de una carpeta del sistema operativo (en nuestro caso, windows, que ejerce de sistema anfitrión). Aunque disponemos de otras opciones que no detallaremos en este guión, lo mejor es usar la primera de las opciones, es decir, disco virtual alojado en un fichero o archivo de nuestro sistema operativo real.





Ahora bien, ese archivo tendrá un tamaño concreto que, por defecto, se fija en 8GB como tamaño mínimo. Parece indicar que si se establece un tamaño escaso se estaría limitando la posibilidad de albergar más contenidos en la máquina virtual. Tal cosa no representa ningún problema pues en la creación del disco virtual es posible indicar si el tamaño será fijo o dinámico (posibilidad de que se pueda incrementar sin pérdida de lo que ya estuviera almacenado). En cualquier caso, para empezar, lo más lógico es indicar un tamaño sensato según el tamaño total del disco duro de nuestro ordenador.

Fijado el tamaño del disco ya tenemos la máquina virtual creada sólo a la espera de proceder con la instalación del sistema operativo. Es importante que previamente se disponga del CD/DVD del sistema operativo a instalar, en este caso el de Ubuntu Linux. Nuestra máquina virtual compartirá los lectores de DVD que existan en la máquina real y cuantas carpetas y/o unidades de disco se deseen compartir.

El proceso de instalación del nuevo sistema operativo sobre la máquina virtual establecida puede ser visto a través de una amplia gama de tutoriales y video-tutoriales disponibles en internet con sólo buscar en google "tutorial instalación de ubuntu en virtualbox". Una estupenda guía se puede encontrar en: `http://www.psychocats.net/ubuntu/virtualbox`

También podemos encontrar un video-tutorial de cómo instalar ubuntu en virtualbox en el siguiente enlace: `http://www.arturogoga.com/2010/05/03/screencast-instalar-ubuntu-10-04-en-windows-maquina-virtual/`

Además, para compartir archivos entre el sistema anfitrion (windows) y el huesped (Ubuntu) podemos observar el siguiente enlace: `http://usemoslinux.blogspot.com/2010/06/como-compartir-carpetas-entre-windows-y.html`

0.5 Introducción a la Interfaz

0.5.1 Identificación en los ordenadores de la ETSIIT

Los ordenadores de las aulas de la ETSIIT disponen de una configuración básica de arranque que difiere del sistema de arranque de cualquier ordenador personal. Una vez puesto en marcha, el ordenador nos instará a que nos identifiquemos con nuestros datos de usuario y contraseña.

Una vez que un estudiante está matriculado en alguna titulación de la Universidad de Granada, se le ha indicado tanto una dirección de correo electrónico con el sufijo `@correo.ugr.es` y datos de usuario y contraseña para acceder al mismo. Del mismo modo, para acceder a los ordenadores de la ETSIIT y, en general, de cualquier ordenador que disponga del mismo protocolo de inicio del sistema, esos mismos datos de usuario y contraseña serán los que se han de introducir al inicio de la identificación de usuario.

Si el sistema respondiese con algún mensaje de error del tipo "el usuario no tiene cuenta" o "datos de usuario incorrectos", se pueden usar las credenciales de la cuenta genérica establecida para tales casos:

Usuario: **generica**
Contraseña: **temporal**

Si es ese el caso, debe informar al administrador de sistemas de tal problema para tratar de solucionarlo de inmediato.

Una vez nos hemos identificado de forma satisfactoria, acto seguido, nos permitirá la elección de qué sistema operativo deseamos iniciar en nuestro puesto de trabajo. En concreto, existen dos sistemas operativos disponibles: Windows y Linux y, dentro de Linux, existe la posibilidad de elegir entre varias distribuciones.

Para las prácticas de esta asignatura elegiremos en todo momento la opción de Linux y, dentro de éste, a modo de recomendación, la última distribución que haya disponible de Ubuntu.

0.5.2 Interfaces comunes

Hoy en día existen muchas interfaces distintas para sistemas Unix y derivados, pero previamente a examinarlas hay que distinguir entre gestor de ventanas ("window manager") y entornos de escritorio ("desktop").

Es posible que en alguna distribución de Linux ya iniciada en los ordenadores de la ETSIIT nos inste a identificarnos nuevamente con los mismos datos de usuario y contraseña reseñados con anterioridad. Alguna de las distribuciones no inicia una interfaz gráfica como la que aparece en la figura 1.5 y en su lugar nos muestra una pantalla negra donde estaremos en disposición de poder interaccionar con el sistema operativo pero sólo a través de las órdenes disponibles. En esos casos, para iniciar el entorno gráfico debemos escribir la orden **`startx`**.

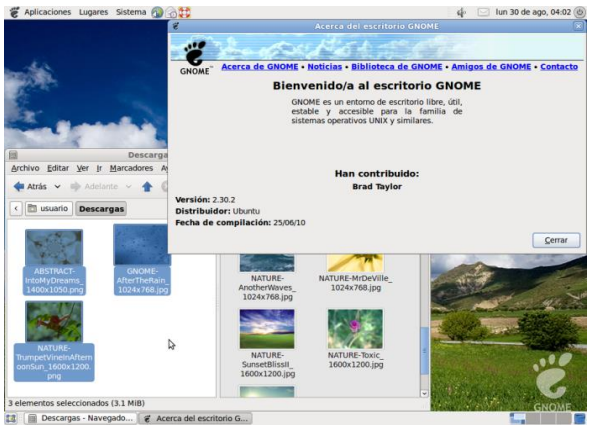


Figura 0.5. Ejemplo de Gnome 2.30.2.

Para terminar la sesión de trabajo y apagar el sistema se recomienda siempre hacerlo de manera natural, es decir, sin tener que pulsar el botón de apagado. Si nos encontramos en una interfaz gráfica basada en un gestor de ventanas podremos acceder a algún lugar del menú y realizar la finalización y apagado del sistema; si nos encontramos con una interfaz en modo texto (fondo negro y caracteres en blanco) hemos de indicarlo mediante la orden **`exit`** y posteriormente realizar el apagado desde el mismo botón de encendido del equipo. No es posible realizar un apagado natural debido a que no se dispone de los privilegios de administrador del sistema.

Los **gestores de ventanas** suelen encargarse de *abrir*, *cerrar*, *minimizar*, *maximizar*, *mover*, *escalar* las ventanas y mantener un listado de las ventanas abiertas. Es también muy común que el gestor de ventanas integre elementos como: el decorador de ventanas, un panel, un visor de escritorios virtuales, iconos y un tapiz.

Estrictamente hablando, un gestor de ventanas para X Windows, no interactúa de forma directa con el hardware de vídeo, ratón o teclado, que son responsabilidad del servidor X.

Las plataformas Windows y Mac OS X ofrecen un gestor de ventanas estandarizado por sus vendedores e integrado en el propio sistema operativo. En cambio el sistema gráfico X Windows, popular en el ámbito de sistemas Unix y derivados, permite al usuario escoger entre varios gestores distintos. Actualmente los más conocidos para sistemas Unix y derivados son: AfterStep, Fvwm / Fvmw2 / Fvmw95, Enlightement, Blackbox, Fluxbox, IceWM, Kwin (KDE), Metacity (GNOME 2), MWM, SawFish, WindowMaker y OLWM / OLVWM. Algunos incluso ofrecen capacidades nativas 3D, tal y como hace Metisse.

Un **entorno de escritorio** es un conjunto de software para ofrecer al usuario de una computadora una interacción amigable y cómoda. El entorno de escritorio es una solución completa de interfaz gráfica de usuario que ofrece iconos, barras de herramientas e integración entre aplicaciones con habilidades como, arrastrar y soltar, entre otras. Actualmente los más conocidos son: GNOME, KDE, CDE, Xfce o LXDE.

La mayoría de las distribuciones incluyen un gestor de ventanas y un entorno de escritorio ambos integrados en una única interfaz.

0.5.3 Gnome vs. KDE

GNOME provee un entorno de escritorio intuitivo y atractivo (según sus creadores) y una plataforma de desarrollo para crear aplicaciones que se integran con el escritorio. El proyecto pone un gran énfasis en la simplicidad, usabilidad y eficiencia. Otros objetivos del proyecto son:

- La libertad para crear un entorno de escritorio que siempre tendrá el código fuente disponible para reutilizarse bajo una licencia de software libre.
- Asegurar la accesibilidad, de modo que pueda ser utilizado por cualquiera, sin importar sus conocimientos técnicos y discapacidad física.
- Hacer que esté disponible en muchos idiomas. Actualmente está siendo traducido a más de 100 idiomas.
- Un ciclo regular de liberaciones y una estructura de comunidad disciplinada.

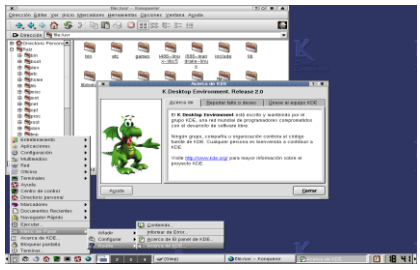


Figura 0.6. Ejemplo de KDE 2.0.

Figura 0.7. Ejemplo de KDE 3.5.

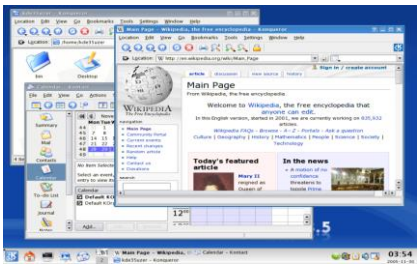


Figura 0.8. Ejemplo de KDE 4.

GNOME suele tener, por tanto, una apariencia estándar en todas sus versiones, manteniendo una barra donde aparece un resumen de las ventanas abiertas o minimizadas, un manejador de escritorios virtuales y una barra con tres menús: "Aplicaciones", "Lugares" y "Sistema". En la figura 1.5 hay un ejemplo de la versión 2.30.2:

Tal y como su nombre indica:

- Aplicaciones: aquí encontramos todas las aplicaciones que están instaladas en el sistema y que generan algún tipo de ventana.
- Lugares: es un acceso rápido al gestor de ficheros.

- Sistema: aquí encontramos programas para administrar y gestionar el sistema, así como la configuración del escritorio y opciones de accesibilidad.

KDE por contra, se define a sí mismo como contemporáneo para estaciones de trabajo Unix y similar a los escritorios de Mac OS X o Windows. KDE se basa en el principio de la personalización; todos los componentes de KDE pueden ser configurados en mayor o menor medida por el usuario.

Cada versión de KDE intenta emular el escritorio de moda de ese año. Así pues, KDE 2.0 se parece mucho a Windows 98, KDE 3.5 a Windows XP y KDE 4 a Windows 7.

Como se puede observar, siempre suele aparecer un botón a izquierda que lanza el menú de aplicaciones y configuración del sistema de una vez.

Ambos utilizan la combinación de teclas rápida ***Alt+F2*** que permite ejecutar un programa escribiendo su nombre. También comparten la combinación ***Alt+Tabulador*** y ***Alt+Shift+Tabulador*** para moverse entre las ventanas.

Ambos pueden integrar efectos 3D en el escritorio mediante ***Compiz–Fusion*** en cualquier estación con una tarjeta gráfica NVIDIA o ATI y también con algunas tarjetas gráficas Intel.

0.5.4 Explorando ficheros y carpetas

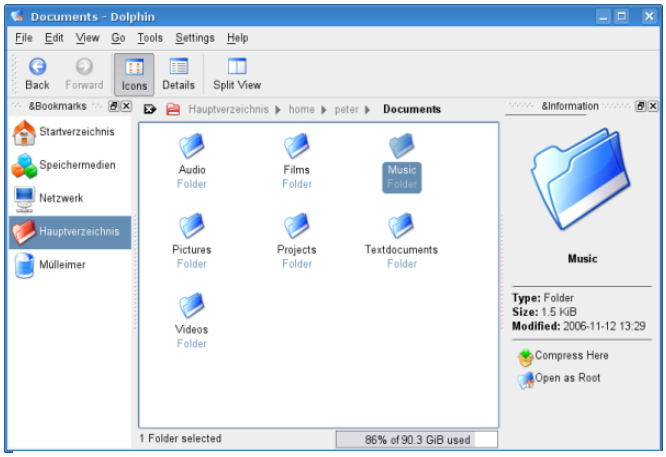


Figura 0.9. Explorador de archivos Dolphin.

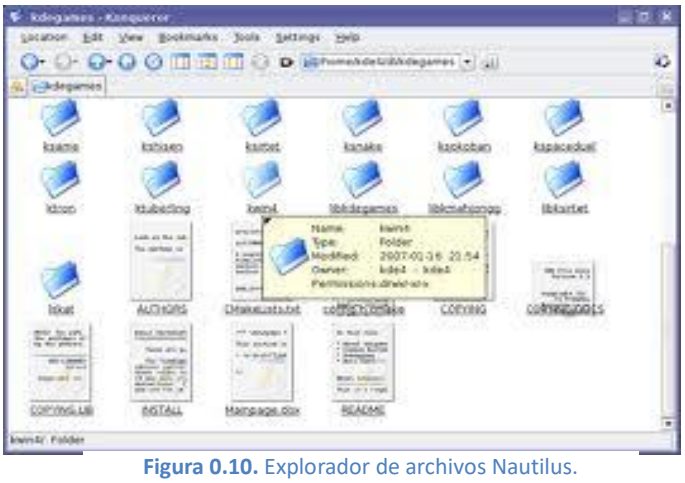


Figura 0.10. Explorador de archivos Nautilus.

GNOME utiliza el explorador de ficheros Nautilus, mientras que KDE utiliza Dolphin (o Konqueror en sus versiones antiguas)

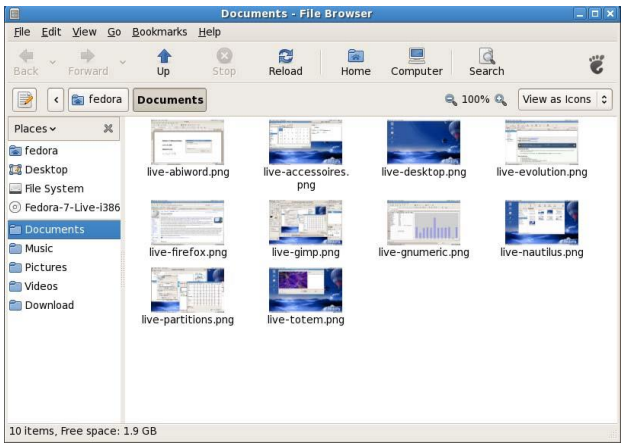


Figura 0.11. Explorador de archivos Konqueror.

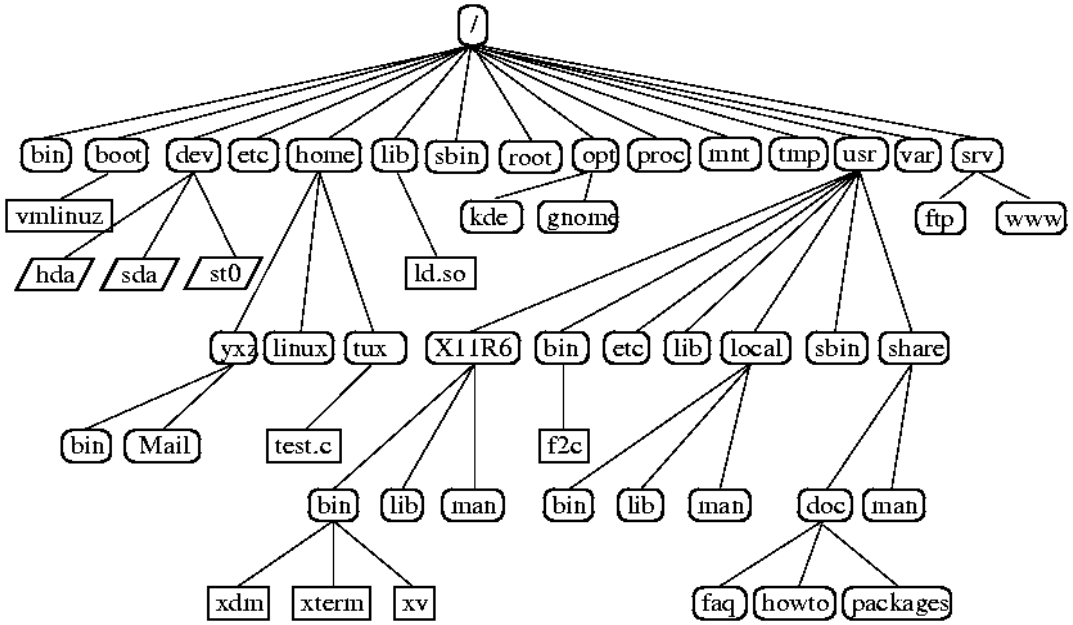


Figura 0.12. Árbol de directorios en un SO basado en UNIX.

Todos ellos tienen una funcionalidad muy similar, permitiendo con el menú contextual (botón derecho) crear carpetas, ficheros, cambiar permisos, etc. Siendo la funcionalidad similar a la de los escritorios de otros sistemas operativos, tales como Windows 98/XP/Vista/7 o Mac OS X.

En la mayoría de los casos, se puede activar un modo para la escritura de rutas URI (identificador uniforme de recurso).

Un URI consta de las siguientes partes:

1. Un identificador o protocolo seguido de "dos puntos": `file:`, `tag:`, `sftp:`, `http:`, etc.

Un elemento jerárquico que identifica la autoridad de nombres precedido por //: `//www.google.com`, `//home/media`, etc.

En el caso de un fichero una URI válida podría ser:

`file:///home/usuario` (nótese la orientación de las barras).

En el caso de los exploradores, si no se introduce el identificador o protocolo ni la // se entiende que hablamos de un fichero o carpeta en nuestro sistema: `/home/usuario`.

Todas las **rutas** (path), entendiendo como ruta una URI en un sistema de ficheros, son un camino en un árbol.

Aquí tenemos un ejemplo de árbol de ficheros típico en las distribuciones Linux:

Las rutas más comunes son las siguientes:

- **sbin**: Programas del sistema que son críticos para el arranque y reservados para el administrador del sistema.
- **bin**: Programas del sistema que no son críticos para el arranque.
- **boot**: Archivos necesarios para el arranque del Sistema Operativo.
- **dev**: Archivos que apuntan a los dispositivos conectados al sistema.
- **etc**: Archivos de configuración del Sistema Operativo.
- **home**: Carpeta principal donde se situarán las carpetas de los distintos usuarios del sistema.
- **lib**: Bibliotecas del sistema.
- **root**: Carpeta del usuario que administra el sistema.

- **opt**: Carpetas con programas añadidos o configurados posteriormente a la instalación del sistema y que no forman parte de la distribución, también se suele usar como carpeta "otros".
- **proc**: Carpeta que contiene archivos con información de dispositivos, también permite configurarlos y procesarlos.
- **mnt**: Unidades adicionales montadas permanentemente en el sistema.
- **media**: Unidades adicionales montadas dinámicamente en el sistema.
- **tmp**: Carpeta con los ficheros temporales, se suele borrar al reiniciar el sistema.
- **usr**: Archivos con programas, librerías, código y documentación de paquetes del sistema.
- **var**: Archivos de log del sistema.
- **srv**: A veces se crea esta carpeta para la gestión de servidores no nativos del sistema o para compartir información con el exterior.

Las rutas pueden ser de dos tipos:

- Absolutas: si comienzan en /
- Relativas: si no comienzan en /

Las rutas relativas tienen que ver con el punto en el que nos encontramos en el árbol actualmente, lo que se suele denominar **directorio actual** *(current directory)*.

Por defecto, el usuario tiene un lugar donde tiene permiso para crear, leer y ejecutar cualquier fichero y carpeta. Este lugar es denominado directorio home, y es el *directorio actual* por defecto del usuario al entrar en el sistema. Todos los exploradores de ficheros permiten volver al directorio home simplemente pulsando en el icono de la casa.

0.5.5 Tipos de archivos y propiedades

MIME (en español "extensiones multipropósito de correo de internet") es una serie de convenciones o especificaciones dirigidas al intercambio a través de Internet de todo tipo de archivos (texto, audio, vídeo, etc.) de forma transparente para el usuario. En KDE y GNOME se utilizan para identificar un fichero y saber con qué programas abrir dicho fichero. Desde Konqueror / Dolphi y Nautilus se pueden configurar los tipos MIME. Suele dividirse entre tipo y subtipo, así pues, text/plain sería texto plano, y podríamos seleccionar un editor de texto cualquiera para abrir ese tipo de ficheros.

Figura 0.13. Propiedades de un archivo o directorio en un SO basado en UNIX.

Normalmente también se suelen tener en cuenta las **extensiones de los ficheros**, aunque normalmente los exploradores Unix suelen mirar el contenido del fichero identificando el tipo del mismo.

Las propiedades de un fichero suelen ser:

- Tamaño: el tamaño que el fichero ocupa en disco.
- Su nombre: el nombre del mismo.
- Permisos: los permisos que se le otorga a un fichero.
- Ejecución: permite que se pueda ejecutar el fichero como si fuera un programa (especialmente útil en guiones [scripts]). También en carpetas permite o deniega el acceso a la misma.
- Lectura: permite leer un fichero o ver el contenido de una carpeta.
- Escritura: impide cambiar de nombre y sobrescribir un fichero, así como alterarlo. En carpetas impide que se pueda crear contenidos en su interior.

Normalmente, en la interfaz suelen estar asociados, de tal forma que se habla de acceder a una carpeta cuando se tienen los permisos de lectura y ejecución sobre la misma.

Los permisos están divididos en tres apartados:

- **Propietario**: permisos para el que creó el fichero.
- **Grupo**: permisos para un grupo al que el usuario pertenece, algunas veces nos deja elegir el grupo siempre que estemos en dicho grupo.
- **Otros**: el resto de usuarios del sistema.

0.5.6 Editores del sistema

Normalmente se suelen dividir los editores en dos tipos: los de modo gráfico y los de modo texto. En editores de tipo texto tenemos algunos muy comunes:

- nano / pico.
- vi.
- emacs.

En editores de tipo gráfico hay muchos, pero los más comunes en sistemas KDE y GNOME son:

- kedit/kwrite.
- kate.
- gedit.

nano/pico es un editor sencillo. Utiliza (al igual que emacs) de forma abusiva la tecla control para el manejo del mismo, la ventaja principal de estos editores (junto con emacs) es que la mayoría de accesos de teclado funcionan también en los terminales del sistema. Algunas combinaciones básicas son:

- Control + A: ir al inicio de la línea [estándar].
- Control + E: ir al final de la línea [estándar].
- Control + F: ir a la derecha un espacio [estándar].
- Control + B: ir a la izquierda un espacio [estándar].
- Control + P: ir arriba un espacio [estándar].
- Control + N: ir abajo un espacio [estándar].
- Control + K: cortar una línea [estándar].
- Control + U: pegar la línea (el estándar es Control + Y).
- Control + O: guarda el fichero.
- Control + W: busca una palabra o frase en un fichero.
- Control + X: salir.
- Control + G: muestra la lista completa de comandos.

Los editores **vi** y **emacs** disponen de modos de trabajo, lo que permite configurar el propio editor para tareas específicas tales como programar, mejorando el rendimiento al escribir código. Son editores complicados de utilizar, pero muy potentes. Las teclas estándar funcionan en emacs pero no en vi, la ventaja fundamental de vi es que permite detectar las órdenes del editor sin ningún tipo de ambigüedad. Emacs puede trabajar tanto en modo texto como en modo gráfico.

Los editores **gedit** y **kedit/kwrite** permiten resaltar la sintaxis del contenido, son sencillos de utilizar, parecidos a cualquier editor de Windows o Mac OS X, pero no son muy potentes. Algunas funcionalidades a destacar son:

- Cortado de texto automático (Text wrapping).
- Mostrar las líneas a la izquierda del margen (Line numbers).
- Señalar la línea donde estamos actualmente (Current line highlighting).
- Ir a una línea específica (Go to specific line).

kwrite permite adicionalmente añadir marcadores para volver de forma sencilla a una línea previamente marcada. Como desventaja principal, no suele detectar la codificación del fichero, por lo que hay que indicarlo de forma manual. El editor **kate** es parecido a kwrite pero con bastante más funcionalidad en cuanto a la edición de código.

Práctica 1: Órdenes básicas de UNIX/Linux

1.1 Introducción

Esta parte se dedica al manejo del Shell de forma interactiva e introduce un conjunto básico de órdenes para comenzar a trabajar con el sistema operativo.

1.2 Objetivos principales

- Manejo de las órdenes básicas de una shell.
- Utilización del manual de uso en línea.
- Manipulación básica de archivos y directorios.

Se presentarán los metacaracteres que nos ayudan a escribir muchas de las órdenes y daremos algunos consejos para su uso. En esta práctica veremos los metacaracteres de nombres de archivos. Se verán las siguientes órdenes:

Órdenes Linux			
man/info/help		**rm**	**more**
touch	**file**	**mkdir/rmdir**	**head/tail**
ls	**cp**	**cd**	**sort**
pwd	**mv**	**cat**	**clear**

Tabla 1.1. Órdenes de la práctica.

1.3 Los intérpretes de órdenes

Un intérprete de órdenes, o shell en la terminología UNIX, está construido como un programa normal de usuario. Esto tiene la ventaja de que podemos cambiar de intérprete de órdenes según nuestras necesidades o preferencias. Existen diferentes shells: Bourne Again Shell (bash), TC Shell (tcsh) y Z Shell (zch). Estos shells no son exclusivos de Linux ya que se distribuyen libremente y pueden compilarse en cualquier sistema Linux. Podemos ver los shell de los que dispone nuestro sistema mirando en el archivo **`/etc/shells`**. El shell, además de ejecutar las órdenes de Linux, tiene sus propias órdenes y variables, lo que lo convierte en un lenguaje de programación. La ventaja que presenta frente a otros lenguajes es su alta productividad –en muchos casos, una tarea escrita en el lenguaje del shell suele tener menos código que si se tuviera que redactar usando un lenguaje de programación de alto nivel como es lenguaje como C-.

Es posible que en las aulas de la ETSIIT, los terminales se inicien con una shell csh (C-Shell). En ese caso, sería necesario escribir la orden **`bash`** o en el encabezado del guion o en el propio terminal para disponer de una shell acorde a la sintaxis del lenguaje de órdenes de la shell que se verá. Algo descriptivo es que con el intérprete de órdenes de tipo C-Shell, el terminal aparece con el prompt `%`, mientras que con la Shell Bash aparecerá información del usuario, nombre del host e información del directorio donde se encuentra, por defecto `~`, que representa el directorio home del usuario conectado y, junto a lo anterior, el símbolo **`$`**.

1.3.1 Orden man

La orden **man** es el paginador del manual del sistema. Las páginas usadas como argumentos al ejecutar `man` suelen ser normalmente nombres de programas, útiles o funciones. La página de manual asociada con cada uno de esos argumentos es buscada y presentada. Si la llamada da también la sección, `man` buscará sólo en dicha sección del manual. Normalmente, la búsqueda se lleva a cabo en todas las secciones de manual disponibles según un orden predeterminado, y sólo se presenta la primera página encontrada, incluso si esa página se encuentra en varias secciones. Para más ayuda escribir en la consola:

```
$ man man
```

man utiliza las siguientes teclas para buscar texto o desplazarse por el mismo:

- Control + F: avanzar página.
- Control + B: retrasar página.
- Control + P: moverse una línea hacia arriba.
- Control + N: moverse una línea hacia abajo.
- `/texto`: busca el texto que se escribe a continuación (incluyendo los huecos en blanco, en su caso) desde la primera línea mostrada en la pantalla en adelante, marcando todas las ocurrencias encontradas.
- `?texto`: busca el texto que se escribe a continuación (incluyendo los huecos en blanco, en su caso) desde la primera línea mostrada en la pantalla hacia atrás, marcando todas las ocurrencias encontradas.
- `n`: siguiente elemento en la búsqueda.
- `N`: elemento previo en la búsqueda.
- `v`: lanza (si es posible) el editor por defecto para editar el fichero que estamos viendo.
- `q`: sale del **man**. También puede ser `Q`, `:q`, `:Q` y `ZZ`.

Nota: Las cuatro primeras pueden realizarse tanto con la letra en mayúscula como en minúscula. Use la orden **man** cuando no sepa las opciones correctas de una orden que necesite para la realización de un ejercicio. Como recomendación general, indicar que es conveniente usar el shell cuando necesitemos hacer algo con muchos archivos, o hacer la misma tarea de forma repetida.

1.3.2 Órdenes Linux relacionadas con archivos y directorios

La tabla siguiente recoge las órdenes más frecuentes relacionadas con archivos y directorios (carpetas):

Órdenes	Descripción
ls [directorio]	Lista los contenidos de un directorio
cd [directorio]	Cambia de directorio de trabajo. Las abreviaciones . y .. se pueden utilizar como referencia de los directorios actual y padre, respectivamente. El símbolo **~** (pulsando a la vez las teclas `AltGr` y `4`) es el directorio HOME del usuario y el símbolo **/** al inicio de un camino es el directorio raíz del sistema.
pwd	Imprime el camino absoluto del directorio actual
mkdir directorio	Crea un directorio a partir del nombre dado como argumento
rmdir directorio	Borra un directorio existente (si está vacío)
cat [archivo(s)]	Orden multipropósito: muestra el contenido de un archivo o varios, concatena archivos, copia un archivo, crea un archivo de texto o muestra los caracteres invisibles de control.
cp archivo1 archivo2	Copia el archivo1 en el archivo2. Si archivo2 no existe, se crea.

mv fuente destino	Renombra archivos (el archivo fuente puede ser archivo o directorio, al igual que el destino) y puede mover de lugar un archivo o directorio
file archivo(s)	Muestra el tipo de archivo dado como argumento
more archivo(s)	Visualiza un archivo fraccionándolo una pantalla cada vez (existen otros paginadores como `page`, `pg`, etc.). Antes de usar esta orden es conveniente usar la orden `file` para comprobar que se trata de un archivo ascii.
rm directorio_archivos	Borra archivos y directorios con contenido
touch archivo(s)	Si existen los archivos dados como argumentos se modifican su fecha y hora. En caso contrario, se crean con la fecha actual del sistema.
clear	Borra el contenido del terminal actual
tail [archivo(s)]	Muestra la parte final del contenido de un archivo dado como argumento. Por defecto muestra 10 líneas.
head [archivo(s)]	Muestra la parte inicial del contenido de un archivo dado como argumento. Por defecto muestra 10 líneas.
sort [archivo(s)]	Ordena, según un criterio elegido, el contenido de los archivos dados como argumentos

Ejercicio 1.1. Cree el siguiente árbol de directorios a partir de un directorio de su cuenta de usuario. Indique también cómo sería posible crear toda esa estructura de directorios mediante una única orden (mire las opciones de la orden de creación de directorios mediante `man mkdir`). Posteriormente realice las siguientes acciones:

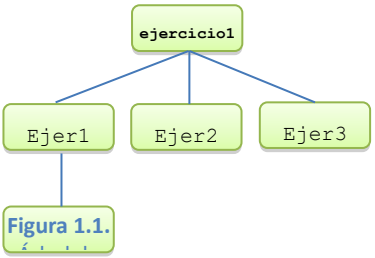


Figura 1.1. Árbol de directorios para el ejercicio 1.

a) En `Ejer1` cree los archivos `arch100.txt`, `filetags.txt`, `practFS.ext` y `robet201.me`.
b) En `Ejer21` cree los archivos `robet202.me`, `ejer11sol.txt` y `blue.me`.
c) En `Ejer2` cree los archivos `ejer2arch.txt`, `ejer2filetags.txt` y `readme2.pdf`.
d) En `Ejer3` cree los archivos `ejer3arch.txt`, `ejer3filetags.txt` y `readme3.pdf`.
e) ¿Podrían realizarse las acciones anteriores empleando una única orden? Indique cómo.

Ejercicio 1.2. Situados en el directorio `ejercicio1` creado anteriormente, realice las siguientes acciones:

a) Mueva el directorio `Ejer21` al directorio `Ejer2`.
b) Copie los archivos de `Ejer1` cuya extensión tenga una `x` al directorio `Ejer3`.
c) Si estamos situado en el directorio `Ejer2` y ejecutamos la orden `ls -la ../Ejer3/*arch*`, ¿qué archivo/s, en su caso, debería mostrar?

Ejercicio 1.3. Si estamos situados en el directorio `Ejer2`, indique la orden necesaria para listar sólo los nombres de todos los archivos del directorio padre.

La orden **ls** (list sources) muestra los archivos contenidos en el directorio que se especifica (si no se especifica nada, tomará como directorio el directorio actual).

ls [-option] … [FILE]...

Algunas de las opciones más corrientes de esta orden son:

-a lista los archivos del directorio actual, incluidos aquellos cuyo nombre comienza con un punto, ".".
-C lista en formato multicolumna.
-l formato largo (ver descripción a continuación).
-r lista en orden inverso.
-R lista subdirectorios recursivamente, además del directorio actual.
-t lista de acuerdo con la fecha de modificación de los archivos.

Por ejemplo, con la opción **-l** veremos los archivos del directorio en formato largo, que da información detallada de los objetos que hay dentro del directorio. Para cada entrada del directorio, se imprime una línea con la siguiente información.

```
$ ls -l
-rw-r--r--  1 x00000 alumnos   23410 Mar 15 2009 programa.c
drwxr-xr-x  2 x00000 alumnos    1024 Aug 11 09:11 practfs
-rwxr-xr-x  1 x00000 alumnos   20250 Mar 15 2009 a.out
```

El significado de la información se irá viendo a lo largo de las sesiones. De momento, comentar dos de los campos informativos:

- **Tipo de objeto:** Un carácter indica el tipo de objeto que representa esa entrada: un – (guión) indica que es un archivo plano o regular; una **d** que es un directorio; la **c** indica que es un archivo de dispositivo orientado a carácter; la **b**, dispositivo orientado a bloques; la letra **l**, archivo de tipo enlace simbólico.

- **Bits de protección:** Parte de la protección en Linux descansa en la protección de los archivos. Cada archivo tiene asociados 12 *bits de protección* que indican qué operaciones podemos realizar sobre él y quien puede hacerlas. La orden **ls** muestra sólo 9 bits a través de una cadena de la forma genérica `rwxrwxrwx` donde cada letra representa un bit (un permiso). La pertenencia a un grupo la establece el administrador del sistema cuando lo crea (en nuestro sistema, todos los alumnos pertenecen al mismo grupo).

Los permisos se indican de la siguiente forma:

Permiso	Archivos	Directorios
r	Lectura (**r**ead)	Se puede listar su contenido
w	Escritura (**w**rite)	Podemos modificarlo
x	Ejecución (e**x**ecute)	Podemos acceder a él
-	No hay permiso	

Existen tres grupos de permisos:

rwx	rwx	rwx
Propietario del archivo (**u**ser)	Grupo de usuarios (**g**roup)	Resto de usuarios (**o**thers)

Algunos ejemplos de las otras órdenes antes citadas:

- Cambiarse al directorio de usuario para trabajar en él (tiene el mismo efecto que **`cd ~`**):

```
$ cd
```

- Copiar el archivo `ejemplo` situado en el directorio `usuario`, que será el nombre de nuestra cuenta de usuario y que está en el mismo nivel que en el que estamos situados, en el directorio `midir`:

```
$ cp ../usuario/ejemplo ../usuario/midir
```

- Asigna el nuevo nombre `megustamas` a un archivo existente, denominado `viejonombre`:

```
$ mv viejonombre megustamas
```

- Creamos un directorio para prácticas de fundamentos del software y borramos un directorio `temporal` que no nos sirve:

```
$ mkdir FS
$ rmdir temporal
```

- Ver el tipo de contenido de uno o varios archivos para, por ejemplo, visualizar el contenido de aquellos que son texto:

```
$ file practica1 programa.c a.out
practica1: ASCII text
programa.c: ISO-8859 c PROGRAM TEXT
a.out:      ELF 32-bit LSB executable, Intel 80386, version 1 …

$ cat practica1 programa.c
```

En ocasiones, podemos por error visualizar un archivo con la orden **`cat`**. Si el archivo contiene caracteres no imprimibles, por ejemplo, un archivo ejecutable, la visualización del mismo suele dejar el terminal con caracteres extraños. Podemos restablecer el estado normal del terminal con la orden siguiente:

```
$ setterm -r
```

Ejercicio 2.4. Liste los archivos que estén en su directorio actual y fíjese en alguno que no disponga de la fecha y hora actualizada, es decir, la hora actual y el día de hoy. Ejecute la orden **`touch`** sobre dicho archivo y observe qué sucede sobre la fecha del citado archivo cuando se vuelva a listar.

Ejercicio 2.5. La organización del espacio en directorios es fundamental para poder localizar fácilmente aquello que estemos buscando. En ese sentido, realice las siguientes acciones dentro de su directorio **`home`** (es el directorio por defecto sobre el que trabajamos al entrar en el sistema):

a) Obtenga en nombre de camino absoluto (pathname absoluto) de su directorio home. ¿Es el mismo que el de su compañero/a?

b) Cree un directorio para cada asignatura en la que se van a realizar prácticas de laboratorio y, dentro de cada directorio, nuevos directorios para cada una de las prácticas realizadas hasta el momento.

c) Dentro del directorio de la asignatura fundamentos del software (llamado **FS**) y dentro del directorio creado para esta práctica, copie los archivos **`hosts`** y **`passwd`** que se encuentran dentro del directorio **`/etc`**.

d) Muestre el contenido de cada uno de los archivos.

1.4 Metacaracteres de archivo

Cuando se pretende aplicar determinadas acciones a un conjunto de archivos y/o directorios cuyos nombres contienen secuencias de caracteres comunes (por ejemplo, `tema1.pdf`, `tema2.pdf` y `tema3.pdf`), es muy útil

poder nombrarlos a través de una expresión que los pudiera identificar a todos (patrón de igualación). El patrón nos permite pasarle a la orden (sustitución de nombres) que vayamos a utilizar una única expresión para poder manipular todo el conjunto. Por ejemplo, para listar los tres archivos anteriores podemos utilizar la orden `ls tema?.pdf`

Metacarácter	Descripción de la función
?	Representa cualquier carácter simple en la posición en la que se indique
*****	Representa cualquier secuencia de cero o más caracteres
[ ]	Designan un carácter o rango de caracteres que representan un carácter simple a través de una lista de caracteres o mediante un rango, en cuyo caso, mostramos el primer y último carácter del rango separados por un guión "-".
{ }	Sustituyen conjuntos de palabras separadas por comas que comparten partes comunes.
~	Se usa para abreviar el camino absoluto (path) del directorio HOME.

Como ejemplo de uso de los metacaracteres de archivos pueden ser los siguientes:

- Igualando un carácter simple con **?:**

```
$ ls -l Sesion.1 Sesion.2 Sesion.3
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.1
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.2
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.3
```

```
$ ls -l Sesion.?
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.1
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.2
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.3
```

- Igualando cero o más caracteres con ***:**

```
$ ls -l Sesion.*
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.1
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.2
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.3
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 Sesion.apendices *.
```

- Igualando cero o más caracteres con **[]:**

```
$ ls -l *.[ch]
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 hebras.c
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 hebras.h
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 pipes.c
```

- Igualando cero o más caracteres con **{}:**

```
$ ls -l {hebras,pipes}.c
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 hebras.c
-rw-r--r-- 1 jrevelle prcicyt 0 oct  7 11:06 pipes.c
```

- Abreviando el directorio home con **~**:

```
$ pwd
/tmp
```

```
$ cd ~/fs
```

```
$ pwd
/home/x01010101/fs
```

Ejercicio 1.6. Situados en algún lugar de su directorio principal de usuario (directorio HOME), cree los directorios siguientes: `Sesion.1`, `Sesion.10`, `Sesion.2`, `Sesion.3`, `Sesion.4`, `Sesion.27`, `Prueba.1` y `Sintaxis.2` y realice las siguientes tareas:

a) Borre el directorio `Sesion.4`
b) Liste todos aquellos directorios que empiecen por `Sesion.` y a continuación tenga un único carácter
c) Liste aquellos directorios cuyos nombres terminen en `.1`
d) Liste aquellos directorios cuyos nombres terminen en `.1` o `.2`
e) Liste aquellos directorios cuyos nombres contengan los caracteres `si`
f) Liste aquellos directorios cuyos nombres contengan los caracteres `si` y terminen en `.2`

Ejercicio 1.7. Desplacémonos hasta el directorio **`/bin`**, genere los siguientes listados de archivos (siempre de la forma más compacta y utilizando los metacaracteres apropiados):

a) Todos los archivos que contengan sólo cuatro caracteres en su nombre.
b) Todos los archivos que comiencen por los caracteres **d**, **f**.
c) Todos los archivos que comiencen por las parejas de caracteres **sa**, **se**, **ad**.
d) Todos los archivos que comiencen por **t** y acaben en **r**.

Ejercicio 1.8. Liste todos los archivos que comiencen por **tem** y terminen por **.gz** o **.zip** :

a) De tu directorio HOME.
b) Del directorio actual.
c) ¿Hay alguna diferencia en el resultado de su ejecución? Razone la respuesta.

Ejercicio 1.9. Muestre del contenido de un archivo regular que contenga texto:

a) Las 10 primeras líneas.
b) Las 5 últimas líneas.

Ejercicio 1.10. Cree un archivo empleando para ello cualquier editor de textos y escriba en el mismo varias palabras en diferentes líneas. A continuación trate de mostrar su contenido de manera ordenada empleando diversos criterios de ordenación.

Ejercicio 1.11. ¿Cómo podría ver el contenido de todos los archivos del directorio actual que terminen en **.txt** o **.c**?

Práctica 2: Permisos y redirecciones

2.1 Objetivos principales

- Modificar los permisos de un archivo.
- Comprender cómo se manejan las entradas y salidas de las órdenes con los operadores de redirección.
- Ver algunas formas de combinar varias órdenes, mediante los metacaracteres sintácticos más usuales.

Además, se verán las siguientes órdenes:

Órdenes Linux			
chmod	**wc**	**echo**	**date**

Tabla 2.1. Órdenes de la práctica.

2.2 Modificación de los permisos de acceso a archivos

En la práctica anterior se han estudiado cuáles son los permisos de acceso a un archivo (lectura, escritura y ejecución) y cómo se pueden conocer dichos permisos (`ls -l`). En este apartado veremos cómo se pueden modificar dichos permisos con la orden `chmod`. Obviamente, se debe ser el propietario del archivo para poder cambiar dichos permisos. Dicha orden tiene dos modos de funcionamiento (sólo estudiaremos el primero de ellos):

- **Simbólico**, para poder cambiar uno o varios de los bits de protección sin modificar el resto.

- **Absoluto**, que cambia todos los permisos, expresándolos como tres cifras en base 8 (octales). En este caso, los valores tanto para el propietario, grupo o resto de usuarios oscilan entre el 0 (ningún privilegio) hasta el 7 (todos los privilegios). Si dicho valor se codifica en binario, se necesitan tres bits que se corresponden con cada permiso (lectura, escritura y ejecución).

En el modo simbólico, se debe indicar primero a qué grupo de usuarios se va a aplicar el cambio con una letra minúscula:

- **u** : propietario
- **g** : grupo
- **o** : resto de usuarios
- **a** : todos los grupos de usuarios

Después, se debe indicar si va a permitir el acceso (+) o se va a denegar el acceso (-). Por último, se indica qué tipo de permiso es el que estamos modificando (r , w , x) y el archivo al que se le van a modificar los permisos. A continuación, se muestran algunos ejemplos de utilización de la orden **chmod**:

```
$ ls -l
-rw-r--r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-r--r--  1 quasimodo alumnos    3410 May 18 2010 ej2
$ chmod g+w ej2
$ ls -l
-rw-r--r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-rw-r--  1 quasimodo alumnos    3410 May 18 2010 ej2
```

```
$ chmod o-r ej1
$ ls -l
-rw-r-----  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-rw-r--  1 quasimodo alumnos    3410 May 18 2010 ej2
$ chmod a+x ej1
$ ls -l
-rwxr-x--x  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-rw-r--  1 quasimodo alumnos    3410 May 18 2010 ej2
```

Si en algún momento se permite el acceso (+) a un permiso que ya estaba activado en el archivo o se quita un permiso (-) que no estaba activado en el archivo, la orden `chmod` no tiene ningún efecto. Es posible combinar varias letras en el apartado de grupo de usuarios para que se aplique a más de uno y también al tipo de permisos. También es posible juntar varios cambios en una única orden `chmod` si los separamos por comas y los cambios pueden aplicarse a más de un archivo. Ejemplos:

```
$ ls -l
-rw-r--r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-r--r--  1 quasimodo alumnos    3410 May 18 2010 ej2
$ chmod og+w ej2
$ ls -l
-rw-r--r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-rw-rw-  1 quasimodo alumnos    3410 May 18 2010 ej2
$ chmod ug-r ej1
$ ls -l
--w----r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-rw-r--  1 quasimodo alumnos    3410 May 18 2010 ej2
$ chmod ug+rw ej1
-rw-rw-r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-rw-r--  1 quasimodo alumnos    3410 May 18 2010 ej2
$ chmod u+x,g-w ej2
-rw-rw-r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rwxr--r--  1 quasimodo alumnos    3410 May 18 2010 ej2
$ chmod g+x ej*
-rw-rwxr--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rwxr-xr--  1 quasimodo alumnos    3410 May 18 2010 ej2
$ chmod 754 ej*
-rwxr-xr--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rwxr-xr--  1 quasimodo alumnos    3410 May 18 2010 ej2
```

Para especificar los permisos en la orden `chmod` tiene la posibilidad de hacerse también de forma numérica en codificación octal. Para más información puede usar la orden `man chmod`.

Ejercicio 2.1. Se debe utilizar solamente una vez la orden **`chmod`** en cada apartado. Los cambios se harán en un archivo concreto del directorio de trabajo (salvo que se indique otra cosa). Cambiaremos uno o varios permisos en cada apartado (independientemente de que el archivo ya tenga o no dichos permisos) y comprobaremos que funciona correctamente:

- Dar permiso de ejecución al "resto de usuarios".
- Dar permiso de escritura y ejecución al "grupo".
- Quitar el permiso de lectura al "grupo" y al "resto de usuarios".
- Dar permiso de ejecución al "propietario" y permiso de escritura el "resto de usuarios".
- Dar permiso de ejecución al "grupo" de todos los archivos cuyo nombre comience con la letra "e". Nota: Si no hay más de dos archivos que cumplan esa condición, se deberán crear archivos que empiecen con "e" y/o modificar el nombre de archivos ya existentes para que cumplan esa condición.

2.3 Metacaracteres de redirección

Dispositivo	Valor
stdin	0
stdout	1
stderr	2

El tratamiento de las entradas y salidas en UNIX/Linux es muy simple ya que todas ellas se tratan como flujos de bytes. Cada programa tendrá siempre un dispositivo[1] de entrada estándar (por defecto, el teclado, identificado con el número 0), un dispositivo de salida estándar (por defecto, la pantalla, identificada con el número 1) y un dispositivo estándar para la salida de errores u otra información (por defecto, también es la pantalla, identificada con el número 2). En general, se maneja el siguiente convenio: Si a una orden que lee o escribe en un archivo, no se le especifica un nombre de archivo, la lectura o escritura se realizará por defecto desde la entrada o en la salida estándar. Por ejemplo, si a la orden `cat` no le pasamos un nombre de archivo, al ejecutarla leerá de la entrada estándar, es decir, de lo que escriba el usuario desde el teclado. Pruébelo (para terminar pulse la combinación de teclas `Ctrl C`).

Los metacaracteres de redirección permiten alterar ese flujo por defecto y, por tanto, redireccionar la entrada estándar desde un archivo, y redirigir tanto la salida estándar como el error estándar hacia archivos, además de poder enlazar la salida de una orden con la entrada de otra permitiendo crear un cauce (*pipeline*) entre varias órdenes. La tabla siguiente muestra los metacaracteres de redirección más usuales:

Metacarácter	Descripción
< nombre	Redirecciona la entrada de una orden para que la obtenga del archivo *nombre*.
> nombre	Redirige la salida de una orden para que la escriba en el archivo *nombre*. Si dicho archivo ya existe, lo sobreescribe.
&> nombre	La salida estándar se combina con la salida de error estándar y ambas se escriben en el archivo *nombre*.
>> nombre	Funciona igual que el metacarácter "**>**" pero añade la salida estándar al final del contenido del archivo *nombre*.
&>> nombre	Igual que el metacarácter "**&>**", pero añadiendo las dos salidas combinadas al final del archivo *nombre*.
2> nombre	Redirige la salida de error estándar a un archivo (sólo funciona en shells de "bash").
|	Crea un cauce entre dos órdenes. La salida de una de ellas se utiliza como entrada de la otra.
|&	Crea un cauce entre dos órdenes utilizando las dos salidas (estándar y error) de una de ellas como entrada de la otra.

2.3.1 Redirección de la entrada estándar (<)

Algunas órdenes toman su entrada de archivos cuyo nombre se pasa como argumento, pero si no se especifica dicho archivo, la lectura se lleva a cabo desde la entrada estándar. Otras órdenes sólo leen de la entrada estándar (como la orden **`mail`**), por lo que si queremos que lean desde un archivo debemos usar el metacarácter de redirección de entrada.

1 Los dispositivos en UNIX/Linux se representan como archivos.

Como ejemplo, obtendríamos el mismo resultado ejecutando las siguientes órdenes:

```
$ cat archivo

$ cat < archivo
```

2.3.2 Redirección de la salida estándar (> , >>)

Las salidas de las órdenes se dirigen normalmente a la pantalla, pudiéndose almacenar en un archivo utilizando los metacaracteres de redirección de salida. Si el nombre del archivo al que se redirecciona la salida no existe, ambos metacaracteres lo crean. La diferencia entre ellos aparece en el caso de que dicho archivo existiera previamente, ya que si usamos "**>**" se borra completamente dicho archivo antes de escribir la salida de la orden mientras que usando "**>>**" no se pierde la información previa que contenía el archivo y se añade la salida de la orden al final del archivo. A continuación, se muestran algunos ejemplos de utilización de estos metacaracteres, suponiendo que inicialmente sólo hay dos archivos en nuestro directorio de trabajo (*notas* y *listado*).

```
$ pwd
/home/users/quasimodo
$ ls
listado  notas
$ ls > temporal
$ ls
listado  notas  temporal
$ cat temporal
listado
notas
$ pwd > temporal
$ cat temporal
/home/users/quasimodo
$ ls >> temporal
$ cat temporal
/home/users/quasimodo
listado
notas
temporal
```

En el siguiente ejemplo, mostramos por pantalla, de todos los archivos del directorio de trabajo que empiezan por la letra "e", el listado (en formato largo) de los dos primeros (para ello se hace uso de la orden **`head -n <file>`**, que muestra las **`n`** primeras líneas del archivo dado como argumento **`<file>`**).

```
$ ls e*
ej1  ej31  ej32  ej4
$ ls -l e* > temporal
$ head -2 temporal
-rw-r--r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-r--r--  1 quasimodo alumnos    3410 May 18 2010 ej31
$ rm temporal
```

Ejercicio 2.2. Utilizando solamente las órdenes de la práctica anterior y los metacaracteres de redirección de salida:

- Cree un archivo llamado *ej31* , que contendrá el nombre de los archivos del directorio padre del directorio de trabajo.
- Cree un archivo llamado *ej32* , que contendrá las dos últimas líneas del archivo creado en el ejercicio anterior.
- Añada al final del archivo *ej32* , el contenido del archivo *ej31* .

2.3.3 Redirección del error estándar (&> , &>>)

Las salidas de las órdenes se dirigen normalmente a la salida estándar. Sin embargo, muchas órdenes escriben mensajes de error o información adicional en otro flujo de bytes que es la salida de error estándar (normalmente, la pantalla). Si se redirige la salida de una orden sólo con el metacarácter ">", los mensajes de error siguen saliendo por la pantalla. Si añadimos el carácter "&" a los metacaracteres de redirección de salida, se redirigen hacia el archivo indicado tanto la salida estándar como el error estándar. También es posible redirigir solamente la salida de error estándar a un archivo se utiliza el metacarácter "2>", aunque sólo funciona en shells de "bash".

A continuación, se muestra un ejemplo de utilización de estos metacaracteres, suponiendo que inicialmente sólo hay dos archivos en nuestro directorio de trabajo `notas` y `listado` y, por tanto, el intentar visualizar el contenido de un archivo inexistente denominado `practica`, origina un mensaje de error.

```
$ ls
listado  notas
$ cat practica
cat: practica: No existe el archivo o el directorio
$ cat practica > temporal
cat: practica: No existe el archivo o el directorio
$ ls
listado  notas  temporal
$ cat temporal

$ cat practica 2> temporal  # también se puede poner cat practica &> temporal
$ cat temporal
cat: practica: No existe el archivo o el directorio
```

2.3.4 Creación de cauces (|)

Los cauces (*pipelines*) son una característica distintiva de UNIX/Linux. Un cauce conecta la salida estándar de la orden que aparece a la izquierda del símbolo | con la entrada estándar que aparece a la derecha de dicho símbolo. Se produce un flujo de información entre ambas órdenes sin necesidad de usar un archivo como intermediario de ambas órdenes.

Como ejemplo de utilización de este mecanismo, a continuación se muestra otra alternativa para realizar el ejemplo presentado al final de la sección 3.2 , que permite implementar el ejercicio de una forma más compacta y sin tener que usar un archivo intermedio.

```
$ ls e*
ej1  ej31  ej32  ej4
$ ls -l e* | head -2
-rw-r--r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
-rw-r--r--  1 quasimodo alumnos    3410 May 18 2010 ej31
```

Ejercicio 2.3. Utilizando el metacarácter de creación de cauces y sin utilizar la orden **`cd`**:

- Muestre por pantalla el listado (en formato largo) de los últimos 6 archivos del directorio **`/etc`**.
- La orden **`wc`** muestra por pantalla el número de líneas, palabras y bytes de un archivo (consulta la orden **`man`** para conocer más sobre ella). Utilizando dicha orden, muestre por pantalla el número de caracteres (sólo ese número) de los archivos del directorio de trabajo que comiencen por los caracteres "e" o "f".

2.4 Metacaracteres sintácticos

Sirven para combinar varias órdenes y construir una única orden lógica. La tabla siguiente muestra los metacaracteres sintácticos más usuales:

Metacarácter	Descripción
`;`	Separador entre órdenes que se ejecutan secuencialmente.
`( )`	Se usan para aislar órdenes separadas por ";" o por "|". Las órdenes dentro de los paréntesis son tratadas como una única orden.
`&&`	Separador entre órdenes, en la que la orden que sigue al metacarácter "&&" se ejecuta sólo si la orden precedente ha tenido éxito (no ha habido errores).
`||`	Separador entre órdenes, en la que la orden que sigue al metacarácter "||" se ejecuta sólo si la orden precedente falla.

2.4.1 Unión de órdenes en la misma línea (;)

El uso del punto y coma permite escribir dos o más órdenes en la misma línea. Las órdenes se ejecutan secuencialmente (como si se hubiesen escrito en líneas sucesivas). En programas del shell permite una asociación visual de órdenes relacionadas (mejora la comprensión del programa y hace que tengas menos líneas). Trabajando de forma interactiva, permite ejecutar varias órdenes sin tener que esperar a que se complete una orden para poder introducir la siguiente. A continuación, se muestra un ejemplo de utilización.

```
$ pwd
/home/users/quasimodo
$ ls -l
-rw-r--r--  1 quasimodo alumnos   23410 Mar 15 2010 ej1
drw-r--r--  1 quasimodo alumnos    3410 May 18 2010 dir1
$ cd dir1 ; ls
programa1
programa2
$ pwd
/home/users/quasimodo/dir1
```

2.4.2 Combinación de órdenes con los paréntesis

Combinando órdenes podremos aislar, cuando nos interese, un cauce, o una secuencia de punto y coma del resto de la línea de órdenes. Lo ilustraremos con un ejemplo utilizando la orden **`date`** (que proporciona la fecha y hora del sistema) y la orden **`pwd`**, que unimos por un punto y coma y creamos un cauce que termina con la orden **`wc`**.

```
$ date
Wed oct 6 10:12:04 WET 2010
$ pwd
/home/users/quasimodo
$ pwd ; date | wc
/home/users/quasimodo
1 6 27
$ (pwd ; date) | wc
2 7 48
```

Como se puede comprobar, el uso de paréntesis produce un resultado diferente, ya que, en el primer caso, la orden **wc** se ejecuta solamente sobre la salida de la orden **date**, mientras que al utilizar paréntesis, es la combinación de las salidas de las 2 órdenes la que se pasa como entrada a la orden **wc**.

2.4.3 Ejecución condicional de órdenes (&& , ||)

El shell proporciona dos metacaracteres que permiten la ejecución condicional de órdenes según el estado de finalización de una de ellas. Separar dos órdenes con **&&** o **||**, provoca que el shell compruebe el estado de finalización de la primera y ejecute la segunda sólo si la primera tiene éxito o falla, respectivamente.

Operador &&: La concatenación de órdenes con && separa dos órdenes de tal forma que la de la derecha sólo se ejecuta si la de la izquierda lo hace correctamente. Por ejemplo, dada la siguiente concatenación de órdenes orden1 && orden2, la orden2 sólo se ejecutará si orden1 terminó sin ningún error.

Operador ||: La concatenación de órdenes con || separa también dos órdenes de forma similar al anterior operador pero, en esta situación, la orden de la derecha sólo se ejecuta cuando la de la izquierda termina de forma incorrecta. Por ejemplo, dada la concatenación de órdenes `orden1 || orden2`, la `orden2` sólo se ejecutará si la `orden1` terminó con algún error.

En los siguientes ejemplos, ilustramos el uso de estos metacaracteres. En ambos casos, la primera orden es la visualización del listado (en formato largo) del archivo *notas* y el resultado varía según esté presente o no dicho archivo en el directorio de trabajo.

```
$ pwd
/home/users/quasimodo
$ ls
listado notas
$ ls -l notas && pwd
-rw-r--r--  1 quasimodo alumnos   3418 Mar 15 2010 notas
/home/users/quasimodo
$ ls -l notas || pwd
-rw-r--r--  1 quasimodo alumnos   3418 Mar 15 2010 notas
$ rm notas
$ ls -l notas && pwd
ls: notas: No existe el archivo o el directorio
$ ls -l notas || pwd
ls: notas: No existe el archivo o el directorio
/home/users/quasimodo
```

No existe ninguna restricción que limite el número de órdenes que aparecen antes del metacarácter **&&** o **||**, pero sólo se evalúa el estado de la última de ellas. Es posible conectar en una misma línea ambos metacaracteres, como vemos en el siguiente ejemplo, en el que, si existe un archivo, queremos que nos muestre el resultado de la orden **wc** aplicada sobre dicho archivo y, en caso de que no exista, nos muestre un mensaje indicativo por pantalla (para eso, utilizamos la orden **echo**).

```
$ ls
```

```
listado
notas
$ ls notas && wc notas || echo "no existe el archivo notas"
notas
86 324 5673 notas
$ rm notas
$ ls notas && wc notas || echo "no existe el archivo notas"
ls: notas: No existe el archivo o el directorio
no existe el archivo notas
```

Ejercicio 2.4. Resuelva cada uno de los siguientes apartados.

a) Cree un archivo llamado `ejercicio1`, que contenga las 17 últimas líneas del texto que proporciona la orden **`man`** para la orden **`chmod`** (se debe hacer en una única línea de órdenes y sin utilizar el metacarácter ";").

b) Al final del archivo `ejercicio1`, añada la ruta completa del directorio de trabajo actual.

c) Usando la combinación de órdenes mediante paréntesis, cree un archivo llamado `ejercicio3` que contendrá el listado de usuarios conectados al sistema (orden **`who`**) y la lista de archivos del directorio actual.

d) Añada, al final del archivo `ejercicio3`, el número de líneas, palabras y caracteres del archivo `ejercicio1`. Asegúrese de que, por ejemplo, si no existiera `ejercicio1`, los mensajes de error también se añadieran al final de `ejercicio3`.

e) Con una sola orden **`chmod`**, cambie los permisos de los archivos `ejercicio1` y `ejercicio3`, de forma que se quite el permiso de lectura al "grupo" y se dé permiso de ejecución a las tres categorías de usuarios.

Práctica 3: Variables, alias, órdenes de búsqueda y guiones

3.1 Objetivos principales

- Conocer el concepto de variable y los tipos de variables que se pueden usar.
- Distinguir entre variables de entorno o globales y variables locales.
- Saber usar los alias y conocer su utilidad.
- Conocer y usar órdenes de búsqueda en archivos y directorios.
- Conocer qué son los guiones del shell (*scripts*) y cómo podemos ejecutarlos.

Además, se verán las siguientes órdenes:

Órdenes Linux			
`set, unset`	`env, printenv`	`declare`	`expr`
`export`	`alias, unalias`	`find`	`grep, fgrep, egrep`
`printf`			

Tabla 3.1. Órdenes de la práctica.

3.2 Variables

Las variables son muy útiles tanto para adaptar el entorno de trabajo al usuario como en la construcción de guiones (o scripts).

3.2.1 Tipos de variables

El bash contempla dos tipos de variables. Las **variables de entorno** o variables globales son aquellas que son comunes a todos los shells. Para visualizarlas puede probar a usar las órdenes **`env`** o **`printenv`.** Por convención, se usan las letras en mayúscula para los nombres de dichas variables.

Ejercicio 3.1: Escriba, al menos, cinco variables de entorno junto con el valor que tienen.

Otro tipo de variables son las llamadas **variables locales**, éstas son sólo visibles en el shell donde se definen y se les da valor. Para ver las variables locales se puede usar la orden **`set`.**

Puede consultar una lista de las variables propias del shell bash comentadas usando la orden de ayuda para las órdenes empotradas:

```
$ help variables
```

3.2.2 Contenido de las variables

Podemos distinguir también las variables según su contenido:

- Cadenas: su valor es una secuencia de caracteres.
- Números: se podrán usar en operaciones aritméticas.
- Constantes: su valor no puede ser alterado.
- Vectores o arrays: conjunto de elementos a los cuales se puede acceder mediante un índice. Normalmente el índice es un número entero y el primer elemento es el 0.

3.2.3 Creación y visualización de variables

Para asignar un valor a una variable bastará con poner el nombre de la variable, un signo igual y el valor que deseamos asignar, que puede ser una constante u otra variable.

¡Cuidado! a cada lado del signo igual no debe haber ningún espacio en blanco. Si delante o detrás del signo igual dejamos un espacio en blanco obtendremos un error, porque lo tomará como si fuera una orden y sus argumentos, y no como una variable. Y, además, el nombre de una variable puede contener dígitos pero no puede empezar por un dígito. Para visualizar el valor de una variable se puede usar la orden **`echo`**.

```
$ numero=1
$ echo $numero
1
```

Para crear variables de tipo vector utilizamos la misma forma de definición pero los elementos del vector se ponen entre paréntesis y separados por espacios. Un ejemplo de creación de un vector es:

```
$ colores=(rojo azul verde)
```

Para acceder a uno de sus elementos:

```
$ echo ${colores[0]}
rojo
$ echo ${colores[1]}
azul
```

Existe una serie de variables especiales y otras que se crean al entrar el usuario. A continuación describiremos algunas de ellas (en el resumen disponible en la plataforma aparece una lista más completa):

Nombre de variable	Descripción
`$BASH`	Contiene la ruta de acceso completa usada para ejecutar la instancia actual de bash.
`$HOME`	Almacena el directorio raíz del usuario; se puede emplear junto con la orden **`cd`** sin argumentos para ir al directorio raíz del usuario.
`$PATH`	Guarda el camino de búsqueda de las órdenes, este camino está formado por una lista de todos los directorios en los que queremos buscar una orden.
`$?`	Contiene el código de retorno de la última orden ejecutada, bien sea una instrucción o un guion.

Tabla 3.2. Variables de entorno predefinidas.

Para borrar una variable se usa la orden **`unset`** junto con el nombre de la variable (o variables) a eliminar.

Si deseamos crear una variable con ciertos atributos, utilizaremos la orden **declare**, cuya sintaxis completa se puede ver con **help declare**. Podemos indicar que una variable es numérica con la opción **-i** y ver los atributos con la opción **-p**:

```
$ declare -i IVA=18
$ declare -p IVA
declare -i IVA="18"
$ declare -i IVA=hola
$ declare -p IVA
declare -i IVA="0"
```

De esta forma cualquier intento de asignar otra cosa diferente de un número a la variable no dará un error. Otros atributos para las variables son: **-r** indica que es de solo lectura, **-a** indica que es una matriz (vector o lista), **-x** indicará que es exportable (ver el sub-apartado siguiente). Estos atributos se pueden mezclar.

3.2.4 Exportar variables

Las variables locales sólo son visibles en el shell actual o en el guion en el que se definen, para emplear estas variables fuera de ellos, debemos exportar su valor para que el sistema lo reconozca, en caso contrario, cuando acaba el guion o el shell las variables toman su valor original:

```
export variable
```

Ejercicio 3.2. Ejecute las órdenes del cuadro e indique qué ocurre y cómo puede resolver la situación para que la variable NOMBRE se reconozca en el shell hijo.

```
$ NOMBRE=FS
$ echo $NOMBRE


$ bash
$ echo $NOMBRE
```

A veces se puede poner en una línea la definición y exportación de una variable:

```
export variable=valor
```

3.2.5 Significado de las diferentes comillas en las órdenes

En el shell bash se puede hacer uso de lo que se denomina *sustitución de órdenes*, que permite la ejecución de una orden, con o sin argumentos, de forma que su salida se trata como si fuese el valor de una variable. La sustitución de órdenes se puede hacer poniendo **$(***orden argumentos***)**, o usando los apóstrofos inversos, es decir `` `orden argumentos` ``, y puede utilizarse en cualquier tipo de expresión, en particular en las expresiones aritméticas que se verán en la práctica siguiente.

Con el ejemplo siguiente se plantean dos expresiones que son equivalentes:

```
$ echo "Los archivos que hay en el directorio son: $(ls -l)"
$ echo "Los archivos que hay en el directorio son: `ls -l`"
```

Ejercicio 3.3: Compruebe qué ocurre en las expresiones del ejemplo anterior si se quitan las comillas dobles del final y se ponen después de los dos puntos. ¿Qué sucede si se sustituyen las comillas dobles por comillas simples?

En los casos anteriores, las comillas dobles se utilizan como mecanismo de *acotación débil* (*weak quotation*), para proteger cadenas desactivando el significado de los caracteres especiales que haya entre ellas, salvo los caracteres `!`, `$`, `\` y `` ` ``, que no quedan protegidos. También se pueden proteger cadenas usando comillas simples como meca-

nismo de *acotación fuerte* (*strong quotation*), aunque en este caso se protegen los caracteres especiales salvo `!`; en consecuencia, las comillas simples serán útiles cuando se quiera proteger una variable o una orden.

Cuando se usan comillas simples dentro de una expresión, por ejemplo, si se deseara imprimir un mensaje como el siguiente: `'En el libro de inglés aparece Peter's cat'`, si se emplea la orden **`echo`**, nos aparecerá en la línea siguiente el carácter **>** que representa una línea de continuación de la orden dada, es decir, es como si no se hubiera completado el mensaje de texto. Para hacerlo correctamente, habría que escapar la comilla de la palabra Peter's con el metacarácter **** y partir la frase en dos partes acotadas con comillas simples como se muestra a continuación:

```
$ echo 'En el libro de inglés aparece Peter's cat'
>
$ echo 'En el libro de inglés aparece Peter'\''s cat'
En el libro de inglés aparece Peter's cat
```

3.2.6 Asignación de resultados de órdenes a variables

Podemos asignar el resultado de una orden a una variable a través del operador ` (comilla invertida). Para ello utilizamos la siguiente declaración:

```
variable=`orden`
```

Por ejemplo, si queremos declarar una variable que se llame `listadearchivos` y que contenga el listado de archivos del directorio actual, podemos hacerlo con la declaración siguiente, donde la variable que se usa será de tipo lista:

```
$ listadearchivos=`ls .`
```

Ahora, la variable contendrá la lista de todos los archivos existentes en el directorio actual.

Es posible que la ejecución de algunas órdenes situadas entre comillas invertidas pudiera producir algún error. Un ejemplo de una situación como esta es cuando se ejecuta la orden `cat <archivo>` y el archivo dado como argumento de la misma no existe. En esta situación, con objeto de depurar el correcto funcionamiento, puede ser útil conocer el estado de la ejecución de la última orden, que valdrá `0` si se ejecutó correctamente, o `1` si hubo algún error, como es el caso del ejemplo que se muestra a continuación:

```
$ cat archivomio
cat: archivomio: No existe el fichero o el directorio
$ echo $?
1
```

Ejercicio 4.4: Pruebe la siguiente asignación:

```
$ numero=$numero+1
$ echo $numero
```

¿Qué ha ocurrido?

Como vemos en el ejemplo anterior, todo se ha convertido en carácter, y no se ha realizado la operación matemática que deseábamos. La solución a este problema viene de la mano de la orden del sistema **`expr`**, con la que podemos evaluar la expresión que le sigue.

```
$ numero=1
$ echo $numero
1
$ numero=`expr $numero + 1`
```

```
$ echo $numero
2
```

Hemos de fijarnos en que la orden **expr** y sus argumentos se encuentran entre apóstrofes inversos, de tal forma que a la variable numero no se asigne la palabra **expr**, sino el resultado de la ejecución de la orden **expr**.

3.3 La orden empotrada printf

La orden echo puede tener comportamientos diferentes según el sistema Unix que utilicemos, por ello es recomendable utilizar la orden printf. La orden empotrada printf (*print format*) imprime un mensaje en la pantalla utilizando el formato que se le especifica. Su sintaxis es:

printf formato [argumentos]

Donde formato es una cadena que describe cómo se deben imprimir los elementos del mensaje. Esta cadena tiene tres tipos de objetos: texto plano, que simplemente se copia en la salida estándar; secuencias de *caracteres de escape*, que son convertidos y copiados en la salida (ver Tabla 3.3); y especificaciones de formato, que se aplican cada una a uno de los argumentos (ver Tabla 3.4).

Secuencia de escape	Acción
\b	Espacio atrás
\n	Nueva línea
\t	Tabulador
\'	Carácter comilla simple
****	Barra invertida
\0n	n = número en octal que representa un carácter ASCII de 8 bits

Tabla 3.3. Algunos códigos de escape.

Código de formato	Representa
%d	Un número con signo
%f	Un número en coma flotante (decimal) sin notación exponencial
%q	Entrecomilla una cadena
%s	Muestra una cadena sin entrecomillar
%x	Muestra un número en hexadecimal
%o	Muestra un número en octal

Tabla 3.4. Algunos códigos de formato.

A continuación veremos algunos ejemplos de la orden. En el primero, se imprime un número en una columna de 10 caracteres de ancho:

```
$ printf "%10d\n" 25
       25
```

Podemos justificar a la izquierda, si utilizamos un número negativo:

```
$ printf "%-10d %-10d\n" 11 12
11      12
```

Si el número que especificamos en el formato es un decimal, la parte entera se interpreta como la anchura de la columna y el decimal como el número mínimo de dígitos:

```
$ printf "%10.3f\n" 15,4
    15,400
```

Podemos convertir un número de octal o hexadecimal a decimal:

```
$ printf "%d %d\n" 010 0xF
8 15
```

Y a la inversa, de decimal a octal/hexadecimal:

```
$ printf "0%o 0x%x\n" 8 15
00 0xf
```

También podemos usar variables como argumentos de la orden **printf**. Por ejemplo, si queremos mostrar la variable IVA, antes declarada, junto con un mensaje explicativo, escribiríamos:

```
$ printf "El valor actual del IVA es del %d\n" $IVA
El valor actual del IVA es del 18
```

3.4 Alias

Los alias se crean con la orden empotrada **alias** y se borran o eliminan con la orden **unalias**. Puedes usar la orden **help** para conocer cuál es la sintaxis de estas dos órdenes.

Los alias son útiles para definir un comportamiento por defecto de una orden o cambiar el nombre de una orden por estar acostumbrado a usar otro sistema. Por ejemplo, los usuarios de Windows pueden estar acostumbrados a usar la orden **dir** para listar el contenido de un directorio, para ellos sería útil usar un alias llamado **dir** que realice esta función:

```
$ alias dir='ls -l'
$ dir
```

Ejecute alias sin argumento y comprobará los alias por defecto que están definidos en su sistema.

Además, dentro de un alias y entre comillas podemos poner varias órdenes separadas por ";" de tal forma que se ejecutarán cada una de ellas secuencialmente.

Para ignorar un alias y ejecutar la orden original (por ejemplo **ls**, y siempre y cuando no haya más de una orden en el alias) se antepone una barra invertida (****) al nombre del alias, de la siguiente forma:

```
$ \ls -l $HOME
```

En algunas distribuciones de Linux, como por ejemplo Ubuntu o Guadalinex, cuando se escribe la orden `ls`, aparece el listado de manera coloreada, es decir, los directorios aparecen en color azul, los archivos con permiso de ejecución aparecen en color verde, etc. En realidad se debe a que ya existe un alias definido con la opción de listar los archivos con la opción de color de forma automática (`ls --color=auto`). Usando la distribución de Linux disponible en el aula de ordenadores, liste los archivos y observe si se muestran con la opción de color, si no se muestra de manera coloreada, defina un alias para que la orden `ls` los muestre.

3.5 Órdenes de búsqueda: find y grep, egrep, fgrep

Es importante disponer de herramientas para realizar búsquedas dentro de los archivos y en la estructura de directorios. Para ello son útiles las siguientes órdenes.

3.5.1 Orden find

Se utiliza para buscar por la estructura de directorios los archivos que satisfagan los criterios especificados. Su formato es el siguiente:

```
find lista-de-directorios [expresiones]
```

Donde `lista-de-directorios` es la lista de directorios a buscar, y las `expresiones` son los operadores que describen los criterios de selección para los archivos que se desea localizar y la acción que se quiere realizar cuando **`find`** encuentre dichos archivos. En las expresiones es posible utilizar los metacaracteres de archivo dados en la práctica nº2.

Los criterios se especifican mediante una palabra precedida por un guion, seguida de un espacio y por una palabra o número entero precedido o no por un **+** o un **-**. Ejemplos de criterios comunes para localizar archivos son:

1. Por el nombre del archivo: se utiliza la opción **`-name`** seguida por el nombre deseado. Este nombre puede incluir la expansión de metacaracteres de archivo debidamente acotados. Por ejemplo:

```
$ find / -name "*.c"
```

2. Por el último acceso: se utiliza la opción **`-atime`** seguida por un número de días o por el número y un signo **+** o **-** delante de él. Por ejemplo:

`-atime 7` busca los archivos a los que se accedió hace 7 días.

`-atime -2` busca los archivos a los que se accedió hace menos de 2 días.

`-atime +5` busca los archivos a los que se accedió hace más de 5 días.

3. Por ser de un determinado tipo: se utiliza la opción **`-type`** seguida de un carácter que indique el tipo de archivo. Se usa la opción **`f`** para referirse a archivos regulares y la opción **`d`** para directorios. En el ejemplo siguiente se muestra la búsqueda de los archivos del directorio actual que sean archivos regulares:

```
$ find . -type f
```

4. Por su tamaño en bloques: se utiliza la opción **`-size`** seguida de un número con o sin signo (**+** o **-**). Si el número va seguido de la letra **`c`** el tamaño dado es en bytes. Por ejemplo: **`-size 100`** busca los archivos cuyo tamaño es de 100 bloques.

Además, se puede negar cualquier operador de selección o de acción utilizando el operador **`!`** que debe ir entre espacios en blanco y antes del operador a negar. Por ejemplo, para buscar los archivos del directorio raíz que no pertenezcan al usuario llamado `pat`:

```
$ find / ! -user pat
```

También se puede especificar un operador u otro utilizando el operador **`-o`**. Este operador conecta dos expresiones y se seleccionarán aquellos archivos que cumplan una de las dos expresiones (y no las dos, como hasta ahora). En el siguiente ejemplo se muestra la búsqueda de los archivos de tamaño igual a 10 bloques o cuyo último acceso (modificación) se haya efectuado hace más de dos días:

```
$ find . -size 10 -o -atime +2
```

Las acciones más comunes son:

-print: visualiza los nombres de camino de cada archivo que se adapta al criterio de búsqueda. Es la opción por defecto. Por ejemplo, para visualizar los nombres de todos los archivos y directorios del directorio actual:

```
$ find . -print
```

-exec: permite añadir una orden que se aplicará a los archivos localizados. La orden se situará a continuación de la opción y debe terminarse con un espacio, un carácter **** y a continuación un **;**. Se utiliza **{}** para representar el nombre de archivos localizados. Por ejemplo:

```
$ find . -atime +100 -exec rm {} \;
```

Eliminará todos los archivos del directorio actual (y sus descendientes) que no han sido utilizados en los últimos 100 días.

-ok: es similar a **-exec**, con la excepción de que solicita confirmación en cada archivo localizado antes de ejecutar la orden.

3.5.2 Órdenes grep, egrep y fgrep

La orden **grep** permite buscar cadenas en archivos utilizando patrones para especificar dicha cadena. Esta orden lee de la entrada estándar o de una lista de archivos especificados como argumentos y escribe en la salida estándar aquellas líneas que contengan la cadena. Su formato es:

```
grep opciones patrón archivos
```

En su forma más sencilla, el patrón puede ser una cadena de caracteres, aunque, como se verá en la siguiente práctica, también pueden usarse expresiones regulares. Por ejemplo, para buscar la palabra `mundo` en todos los archivos del directorio actual usaremos:

```
$ grep mundo *
```

Algunas opciones que se pueden utilizar con la orden **grep** son:

- **-x** localiza líneas que coincidan totalmente, desde el principio hasta el final de línea, con el patrón especificado.
- **-v** selecciona todas las líneas que no contengan el patrón especificado.
- **-c** produce solamente un recuento de las líneas coincidentes.
- **-i** ignora las distinciones entre mayúsculas y minúsculas.
- **-n** añade el número de línea en el archivo fuente a la salida de las coincidencias.
- **-l** selecciona sólo los nombres de aquellos archivos que coincidan con el patrón de búsqueda.
- **-e** especial para el uso de múltiples patrones e incluso si el patrón comienza por el carácter (-).

Existen dos variantes de **grep** que optimizan su funcionamiento en casos especiales. La orden **fgrep** acepta sólo una cadena simple de búsqueda en vez de una expresión regular. La orden **egrep** permite un conjunto más complejo de operadores en expresiones regulares. Usando **man** comprueba las diferencias entre estas tres órdenes.

3.6 Guiones

Un *guion* del shell (script o programa shell) es un archivo de texto que contiene órdenes del shell y del sistema operativo. Este archivo es utilizado por el shell como guía para saber qué órdenes ejecutar.

Siguiendo la similitud de ejemplos de lenguajes de programación de alto nivel, comencemos con el típico ejemplo `"Hola Mundo"` para mostrar dicho mensaje mediante un guion o script bash. Para ello, abriremos un editor de textos ascii (vi, kedit, gedit, emacs, xemacs, ...) y escribimos lo siguiente:

```
echo "Hola Mundo"
```

Mediante el editor guardamos este documento con el nombre `holamundo`; para mostrar el resultado de su ejecución nos vamos al terminal y escribimos lo siguiente:

```
$ bash holamundo
```

Esta orden, lanza un shell bash y le indica que lea las órdenes del guion de prueba, en lugar de usar el propio terminal. De esta forma podemos automatizar procesos al liberar al usuario de estar escribiendo las órdenes repetidamente.

Para tratar de ilustrar las diferentes invocaciones de variables con comillas simples, dobles y con la barra invertida, crearemos un documento que denominaremos `imprimevar` con las siguientes órdenes:

```
variable=ordenador
printf "Me acabo de comprar un $variable\n"
printf 'Me acabo de comprar un $variable\n'
printf "Me acabo de comprar un \$variable\n"
```

Una vez guardado el documento anterior, podremos invocarlo mediante `bash imprimevar` y tendremos el siguiente resultado tras su ejecución:

```
$ bash imprimevar
Me acabo de comprar un ordenador
Me acabo de comprar un $variable
Me acabo de comprar un $variable
```

Otro ejemplo que podemos mostrar es listar los archivos del directorio del usuario. Para ello, creamos un archivo de texto que contenga las siguientes líneas y que denominaremos `prueba`:

```
printf "El directorio $HOME contiene los siguientes archivos:\n"
ls $HOME
```

La invocación del guión anterior sería mediante la orden `bash prueba`, sin embargo, es posible simplificar el proceso de invocación de un guion para que, en lugar de hacerlo explícitamente con la palabra bash, podamos poner dentro del propio archivo el tipo de shell que se debe utilizar aprovechando las facilidades que nos ofrece el sistema operativo.

Para ello, debemos poner siempre en la primera línea del archivo los símbolos **#!** seguidos del nombre del programa del tipo de shell que deseamos ejecutar, para nuestro caso, `/bin/bash`. Con esto, nuestro ejemplo anterior quedaría:

```
#!/bin/bash
printf "El directorio $HOME contiene los siguientes archivos:\n"
ls $HOME
```

Ahora, podemos hacer nuestro archivo ejecutable (`chmod +x prueba`) con lo que la ejecución sería:

```
$ ./prueba
```

Aclaración: Hemos antepuesto **`./`** al nombre de la orden por la siguiente razón: tal y como podemos observar, la variable **`$PATH`** (que contiene la lista de directorios donde el sistema busca las órdenes que tecleamos) no contiene nuestro directorio de trabajo, por lo que debemos indicarle al shell que la orden a ejecutar está en el directorio actual (.).

Nombre	Descripción
`$0`	Nombre del guion o script que se ha llamado. Sólo se emplea dentro del guion.
`$1 .. $9` **`${n}, n>9`**	Son los distintos argumentos que se pueden facilitar al llamar a un guion. Los nueve primeros se referencian con **`$1,$2,...,$9`**, y a partir de ahí es necesario encerrar el número entre llaves, es decir, **`${n}`**, para **`n>9`**.
`$*`	Contiene todos los argumentos que se le han dado. Cuando va entre comillas dobles es equivalente a **"`$1 $2 $3 … $n`"**.
`$@`	Contiene todos los argumentos que se le han dado. Cuando va entre comillas dobles es equivalente a **"`$1`" "`$2`" … "`$n`"**.
`$#`	Contiene el número de argumentos que se han pasado al llamar al guion.
`${arg:-val}`	Si el argumento tiene valor y es no nulo, continua con su valor, en caso contrario se le asigna el valor indicado por *val*.
`${arg:?val}`	Si el argumento tiene valor y es no nulo, sustituye a su valor; en caso contrario, imprime el valor de *val* y sale del guion. Si *val* es omitida, imprime un mensaje indicando que el argumento es nulo o no está asignado.

Tabla 3.5. Variables de entorno definidas para los argumentos de un guion.

Como se ve en la tabla anterior, las variables numéricas nos permiten pasar argumentos a un guion para adaptar su comportamiento, son los *parámetros* del guion. Se puede pasar cualquier número de argumentos, pero por compatibilidad con versiones anteriores del shell que sólo permitía del 0 al 9, los argumentos por encima del 9 se suelen ponen entre llaves, por ejemplo, **`${12}`**[2].

Ahora podemos adaptar el guion `prueba`, creado anteriormente, para que nos muestre el contenido de cualquier directorio:

```
#!/bin/bash
printf "El directorio $1 contiene los siguientes archivos:\n"
ls $1
```

De esta forma, si queremos ver el contenido del directorio `/bin` solo debemos ejecutar:

[2] Si un argumento va asociado a nombres de archivos o directorios y se usan los metacaracteres ya conocidos, se sustituiría el argumento por tanto archivos/directorios que cumplieran el patrón (por ejemplo, dado un directorio que contiene los archivos `ar1.txt` y `ar2.txt`, si un argumento es `*.txt`, lo consideraría como `ar1.txt ar2.txt`). Por lo tanto, en lugar de un argumento pasaría a tener 2, es decir, `$1=ar1.txt` y `$2=ar2.txt`.

```
$ ./prueba /bin
```

Otro ejemplo sencillo de guion bash sería el de realización de una copia en otro directorio de todos los archivos y subdirectorios del directorio home de un usuario. Al igual que antes, se abre el editor con el que más cómodos nos sintamos y escribimos lo siguiente:

```
#!/bin/bash
printf "Haciendo copia de seguridad en $HOME...\n"
cp -r $HOME/* /tmp/backupuser/
printf "Copia realizada\n"
```

Guardamos el documento anterior con el nombre `mybackup` y ya podremos realizar cuando deseemos una copia de seguridad completa de nuestro directorio `$HOME` (suponemos que el directorio `backupuser` ya está creado previamente).

Ejercicio 3.5. Construya un guion que acepte como argumento una cadena de texto (por ejemplo, su nombre) y que visualice en pantalla la frase `Hola` y el nombre dado como argumento.

Ejercicio 3.6. Varíe el guion anterior para que admita una lista de nombres.

Por otra parte, aunque se verá con mayor detalle más adelante en la sección 6.2 Características de depuración en bash, pág. 75), a medida que incluyamos órdenes a un guion es posible que a la hora de ejecutarlo puedan aparecer errores sintácticos. Para corregir dichos errores basta con leer el texto del mensaje que nos muestra en el terminal, observar el número de la línea en la que se indica la localización del error y si tras su análisis no se consigue solucionarlo, será necesario realizar una ejecución depurada del mismo. Para ello, podemos ejecutar nuestro guion con la orden **`bash`** y empleando una de las siguientes opciones:

- **`-n`**: Chequea errores sintácticos pero sin ejecutar el guion.
- **`-v`**: Visualiza cada orden del guion antes de ejecutarla.
- **`-x`**: Actúa igual que `-v` sólo que sustituyendo, en su caso, las variables por los valores que tienen en ese instante.

Supongamos el siguiente ejemplo de guion que contiene un error sintáctico:

```
#!/bin/bash
persona=Antonio
echo "Buenas tardes $persona
```

Como se puede apreciar, en la línea 3 falta cerrar la doble comilla del texto de la orden `echo`. Al ejecutar el guion con las siguientes opciones podremos ver el resultado:

```
$ bash -n ejemplo
ejemplo: línea 3: EOF inesperado mientras se buscaba un `" coincidente
ejemplo: línea 4: error sintáctico: no se esperaba el final del fichero

$ bash -v ejemplo
#!/bin/bash
persona=Antonio
echo "Buenas tardes $persona
ejemplo: línea 3: EOF inesperado mientras se buscaba un `" coincidente
ejemplo: línea 4: error sintáctico: no se esperaba el final del fichero

$ bash -x ejemplo
+ persona=Antonio
ejemplo: línea 3: EOF inesperado mientras se buscaba un `" coincidente
ejemplo: línea 4: error sintáctico: no se esperaba el final del fichero
```

Para una correcta asimilación de todo ello, se recomienda probar las opciones anteriores una vez corregido el error.

3.6.1 Normas de estilo

Entre las normas que se dan a la hora de escribir un guion, se indica que es buena costumbre comentarlo para conocer siempre quién lo ha escrito, en qué fecha, qué hace, etc. Para ello, utilizaremos en símbolo "**#**", bien al inicio de una línea o bien tras una orden. Por ejemplo, nuestros guiones pueden empezar:

```
#!/bin/bash
# Titulo:      prueba
# Fecha:       5/10/2011
# Autor:       Profesor de FS
# Versión:     1.0
# Descripción: Guion de prueba para la práctica 4
# Opciones: Ninguna
# Uso: prueba directorio

printf "El directorio $1 contiene los siguientes archivos:\n"
ls  $1      # lista los archivos del directorio que se le pase como argumento
```

Podemos encontrar más normas de estilo en el documento "Bash Style Guide and Coding Standard" de Fritz Mehner, 2009, disponible en http://lug.fh-swf.de/vim/vim-bash/StyleGuideShell.en.pdf.

También, la Free Software Foundation tiene disponible una serie de guías para escribir software GNU en las que se describe la forma estándar en la que deben operar las utilidades Unix. Está accesible en http://www.gnu.org/prep/standards/. Por ejemplo, un guion debe al menos soportar dos opciones `-h` (o `--help`) para la ayuda y `--version`) para indicar la versión, nombre, autor, etc.

Tomando como referencia el ejemplo de la copia de seguridad visto anteriormente se podría modificar el guion con un argumento que indique el destino de los archivos que se desean copiar.

El guion quedaría como se muestra en el siguiente ejemplo:

```
#!/bin/bash
# Titulo:      mybackup
# Fecha:       5/10/2011
# Autor:       Profesor de FS
# Versión:     1.0
# Descripción: Realiza una copia de seguridad de los archivos del usuario
#              en un directorio dado como argumento.
# Opciones: Ninguna
# Uso: mybackup destino

printf "Haciendo copia de seguridad de $HOME...\n"
cp -r $HOME/* $1
printf "Copia realizada\n"
```

Ejercicio 3.7. Cree tres variables llamadas `VAR1`, `VAR2` y `VAR3` con los siguientes valores respectivamente "hola", "adios" y "14".

a) Imprima los valores de las tres variables en tres columnas con 15 caracteres de ancho.
b) ¿Son variables locales o globales?
c) Borre la variable VAR2.
d) Abra otra ventana de tipo terminal, ¿puede visualizar las dos variables restantes?
e) Cree una variable de tipo vector con los valores iniciales de las tres variables.
f) Muestre el segundo elemento del vector creado en el apartado e.

Ejercicio 3.8. Cree un alias que se llame **`ne`** (nombrado así para indicar el número de elementos) y que devuelva el número de archivos que existen en el directorio actual. ¿Qué cambiaría si queremos que haga lo mismo pero en el directorio home correspondiente al usuario que lo ejecuta?

Ejercicio 3.9. Indique la línea de orden necesaria para buscar todos los archivos a partir del directorio home de usuario (`$HOME`) que tengan un tamaño menor de un bloque. ¿Cómo la modificaría para que además imprima el resultado en un archivo que se cree dentro del directorio donde nos encontremos y que se llame `archivosP`?

Ejercicio 3.10. Indique cómo buscaría todos aquellos archivos del directorio actual que contengan la palabra "ejemplo".

Ejercicio 3.11. Complete la información de `find` y `grep` utilizando para ello la orden `man`.

Ejercicio 3.12. Indique cómo buscaría si un usuario dispone de una cuenta en el sistema.

Ejercicio 3.13. Indique cómo contabilizar el número de ficheros de la propia cuenta de usuario que no tengan permiso de lectura para el resto de usuarios.

Ejercicio 3.14. Modifique el ejercicio 8 de forma que, en vez de un alias, sea un guion llamado **numE** el que devuelva el número de archivos que existen en el directorio que se le pase como argumento.

Práctica 4: Expresiones con variables y expresiones regulares

4.1 Objetivos principales

- Distinguir entre operadores aritméticos y relacionales para definir expresiones con variables.
- Conocer operadores de consulta de archivos y algunas órdenes para utilizarlos.
- Conocer el concepto de expresión regular y operadores para expresiones regulares.
- Saber utilizar distintos tipos de operadores con las órdenes **`find`** y **`grep`**.

Además, en esta práctica se verán las siguientes órdenes:

Órdenes Shell Bash					
`$(( … ))`	**`$[ … ]`**	**`bc`**	**`let`**	**`test`**	**`if/else`**

Tabla 4.1. Órdenes de la práctica.

4.2 Expresiones con variables

Como se vio en la práctica anterior, las variables son muy útiles tanto para adaptar el entorno de trabajo al usuario como en la construcción de guiones (o scripts). Pero en muchas ocasiones esas variables no se tratan de manera independiente, sino que se relacionan unas con otras, como puede ser mediante la orden **`expr`** que también se vio en esa práctica, o mediante expresiones aritméticas.

El shell bash ofrece dos posibles sintaxis para manejar expresiones aritméticas haciendo uso de lo que se denomina *expansión aritmética*, o *sustitución aritmética*, que evalúa una expresión aritmética y sustituye el resultado de la expresión en el lugar donde se utiliza. Ambas posibilidades son:

`$(( … ))`

`$[ … ]`

En estos casos, lo que se ponga en lugar de los puntos suspensivos se interpretará como una expresión aritmética, no siendo necesario dejar huecos en blanco entre los paréntesis más internos y la expresión contenida en ellos, ni entre los corchetes y la expresión que contengan. Además, hay que tener en cuenta que las variables que se usen en una expresión aritmética no necesitan ir precedidas del símbolo **`$`** para ser sustituidas por su valor, aunque si lo llevan no será causa de error, y que cualquier expresión aritmética puede contener otras expresiones aritméticas, es decir, las expresiones aritméticas se pueden anidar.

Por ejemplo, la orden **`date`**, que permite consultar o establecer la fecha y la hora del sistema, y que admite como argumento **`+%j`** para conocer el número del día actual del año en curso, puede utilizarse para saber cuántas semanas faltan para el fin de año:

```
$ echo "Faltan $(( (365 - $(date +%j)) / 7 )) semanas hasta el fin de año"
```

Ejercicio 4.1: Utilizando una variable que contenga el valor entero 365 y otra que guarde el número del día actual del año en curso, realice la misma operación del ejemplo anterior usando cada una de las diversas formas de cálculo comentadas hasta el momento, es decir, utilizando **`expr`**, **`$(( … ))`** y **`$[ … ]`**.

4.2.1 Operadores aritméticos

El shell bash considera, entre otros, los operadores aritméticos que se dan en la Tabla 4.2.

Operador	Descripción
+ -	Suma y resta, o más unario y menos unario.
*** / %**	Multiplicación, división (truncando decimales), y resto de la división.
******	Potencia.
++	Incremento en una unidad. Puede ir como prefijo o como sufijo de una variable: si se usa como prefijo (`++variable`), primero se incrementa la variable y luego se hace lo que se desee con ella; si se utiliza como sufijo (`variable++`), primero se hace lo que se desee con la variable y después se incrementa su valor.
--	Decremento en una unidad. Actúa de forma análoga al caso anterior, pudiendo usarse como prefijo o como sufijo de una variable (`--variable` o `variable--`).
( )	Agrupación para evaluar conjuntamente; permite indicar el orden en el que se evaluarán las subexpresiones o partes de una expresión.
,	Separador entre expresiones con evaluación secuencial.
=	`x=expresión`, asigna a `x` el resultado de evaluar la `expresión` (no puede haber huecos en blanco a los lados del símbolo "=");
+= -=	`x+=y` equivale a `x=x+y`; `x-=y` equivale a `x=x-y`;
***= /=**	`x*=y` equivale a `x=x*y`; `x/=y` equivale a `x=x/y`;
%=	`x%=y` equivale a `x=x%y`.

Tabla 4.2. Operadores aritméticos.

Ejercicio 4.2: Realice las siguientes operaciones para conocer el funcionamiento del operador de incremento como sufijo y como prefijo. Razone el resultado obtenido en cada una de ellas:

```
$ v=1
$ echo $v
$ echo $((v++))
$ echo $v
$ echo $((++v))
$ echo $v
```

Ejercicio 4.3: Utilizando el operador de división, ponga un caso concreto donde se aprecie que la asignación abreviada es equivalente a la asignación completa, es decir, que `x/=y` equivale a `x=x/y`.

En el resultado del cálculo de expresiones aritméticas, bash solamente trabaja con números enteros, por lo que si se necesitase calcular un resultado con decimales, habría que utilizar una forma alternativa, como puede ser la ofrecida por la orden **`bc`**, cuya opción **`-l`**, letra "ele", permite hacer algunos cálculos matemáticos (admite otras posibilidades que pueden verse mediante **`man`**).

El ejemplo siguiente ilustra el uso de la orden **`bc`** para realizar una división con decimales:

```
$ echo 6/5|bc -l
```

Ejercicio 4.4: Compruebe qué ocurre si en el ejemplo anterior utiliza comillas dobles o simples para acotar todo lo que sigue a la orden **echo**. ¿Qué sucede si se acota entre comillas dobles solamente la expresión aritmética que se quiere calcular?, ¿y si se usan comillas simples?

Ejercicio 4.5: Calcule con decimales el resultado de la expresión aritmética `(3-2)/5`. Escriba todas las expresiones que haya probado hasta dar con una respuesta válida. Utilizando una solución válida, compruebe qué sucede cuando la expresión aritmética se acota entre comillas dobles; ¿qué ocurre si se usan comillas simples?, ¿y si se ponen apóstrofos inversos?

4.2.2 Asignación y variables aritméticas

Otra forma de asignar valor a una variable entera es utilizar la orden **let** de la shell bash. Aunque esta orden se usa para evaluar expresiones aritméticas, en su forma más habitual su sintaxis es:

```
let variableEntera=expresión
```

Como en otras asignaciones, a ambos lados del signo igual (=) no debe haber espacios en blanco y `expresión` debe ser una expresión aritmética.

Ejemplo: Compruebe el resultado de cada una de las asignaciones siguientes:

```
$ let w=3+2
$ let w='3 + 2'
$ let w='(4+5)*6'
$ let "w=4+5*6"
$ let w=4+5*6
$ y=7
$ let w=y%5                    (esta orden es equivalente a: let w=$y%5)
```

Como habrá observado en el ejemplo anterior, las dos primeras asignaciones producen el mismo resultado, a pesar de que en la segunda hay espacios en blanco. Por el contrario, las asignaciones tercera y cuarta no dan el mismo resultado debido al uso o no de paréntesis. Las asignaciones cuarta y quinta son equivalentes, y las dos últimas ponen de manifiesto que en la expresión pueden intervenir otras variables.

Hemos de indicar que `(( <expresión> ))` equivale a la orden `let` y presenta ventajas como por ejemplo a la hora de hacer comparaciones numéricas para usarlas en ejecuciones condicionales:

```
$ a=10
$ ((a<10))
$ echo $?
1
$ ((a==10))
$ echo $?
0
$ if let 'a<10'; then echo "es menor"; else echo "es mayor o igual"; fi
es mayor o igual
```

Ejercicio 4.6: Consulte la sintaxis completa de la orden **let** utilizando la orden de ayuda para las órdenes empotradas (**help let**) y tome nota de su sintaxis general.

Ejercicio 4.7: Con la orden **let** es posible realizar asignaciones múltiples y utilizar operadores que nosotros no hemos mencionado anteriormente. Ponga un ejemplo de asignación múltiple y, por otra parte, copie en un archivo el orden en el que se evalúan los operadores que admite. Apóyese a través de la ayuda que ofrece **help let**.

Ejercicio 4.8: Probad los siguientes ejemplos y escribir los resultados obtenidos con la evaluación de expresiones

```
echo ejemplo1
valor=6
if [ $valor = 3 ]; then echo si; else echo no; fi
echo $valor

echo ejemplo2
valor=5
if [ $valor = 3 ] && ls; then echo si; else echo no; fi
echo $valor

echo ejemplo3
valor=5
if [ $valor = 5 ] && ls; then echo si; else echo no; fi
echo $valor

echo ejemplo4
valor=6
if ((valor==3)); then echo si; else echo no; fi
echo $valor

echo ejemplo5
valor=5
if ((valor==3)) && ls; then echo si; else echo no; fi
echo $valor

echo ejemplo6
valor=5
if ((valor==5)) && ls; then echo si; else echo no; fi
echo $valor

echo ejemplo7
echo $((3>5))
echo $?

echo ejemplo8
((3>5))
echo $?

echo ejemplo9
if ((3>5)); then echo 3 es mayor que 5; else echo 3 no es mayor que 5; fi
```

4.2.3 Operadores relacionales

A veces es necesario poder relacionar dos expresiones aritméticas, `A` y `B`, o negar una expresión aritmética, de forma que se pueda evaluar si se da o no cierta relación. La evaluación de una relación entre expresiones tomará finalmente un valor numérico, de manera que el 1 representa una evaluación "verdadera" (*true*), mientras que el 0 indica que la evaluación ha sido "falsa" (*false*). Observe que esta forma de evaluar resulta un poco discordante respecto a lo que sucede cuando se evalúa la variable `$?` que se mencionaba en la práctica anterior.

A continuación, en Tabla 4.3, se pueden ver diferentes operadores relacionales admitidos en el shell bash.

Operador	Descripción: el resultado se evalúa como "verdadero" -*true*- si ... (en otro caso sería "falso" -*false*-)
`A = B` | `A == B` | `A -eq B`	`A` es igual a `B`.
`A != B` | `A -ne B`	`A` es distinta de `B`.
`A < B` | `A -lt B`	`A` es menor que `B`.
`A > B` | `A -gt B`	`A` es mayor que `B`.
`A <= B` | `A -le B`	`A` es menor o igual que `B`.
`A >= B` | `A -ge B`	`A` es mayor o igual que `B`.
`! A`	`A` es falsa; representa al operador NOT (negación lógica).
`A && B`	`A` es verdadera y `B` es verdadera; es el operador AND (conjunción lógica).
`A || B`	`A` es verdadera o `B` es verdadera; es el operador OR (disyunción lógica).

Tabla 4.3. Operadores relacionales.

Por otra parte, cabe observar que existen otros operadores aparte de los mencionados aquí.

Por ejemplo, se puede comprobar que la expresión `(8>3)&&(9<5)` es falsa, ya que la primera parte de ella es verdadera, pero la segunda es falsa:

```
$ echo $[$[8>3]&&$[9<5]]
0
$ echo $[8>3] y $[9<5]
1 y 0
```

Ejercicio 4.9: Haciendo uso de las órdenes conocidas hasta el momento, construya un guion que admita dos parámetros, que compare por separado si el primer parámetro que se le pasa es igual al segundo, o es menor, o es mayor, y que informe tanto del valor de cada uno de los parámetros como del resultado de cada una de las evaluaciones mostrando un 0 o un 1 según corresponda.

4.2.4 Operadores de consulta de archivos

A veces es necesario conocer características específicas de los archivos o directorios para saber cómo tratarlos. En Tabla 4.4 se pueden ver algunos operadores utilizados para la comprobación de características de archivos y directorios.

Para aplicar los operadores de consulta de archivos haremos uso de dos órdenes nuevas, **`test`** e **`if`**, aunque la segunda de estas órdenes la trataremos en un apartado posterior.

La sintaxis de la orden **`test`** es:

`test` *`expresión`*

Esta orden evalúa una expresión condicional y da como salida el estado 0, en caso de que *`expresión`* se haya evaluado como verdadera (*true*), o el estado 1, si la evaluación ha resultado falsa (*false*) o se le dio algún argumento no válido.

Operador	Descripción: el resultado se evalúa como "verdadero" -*true*- si ... (en otro caso sería "falso" -*false*-)
-b *archivo*	*archivo* existe y es un dispositivo de bloques.
-c *archivo*	*archivo* existe y es un dispositivo de caracteres.
-d *archivo*	*archivo* existe y es un directorio.
-e *archivo*	*archivo* existe.
-f *archivo*	*archivo* existe y es un archivo plano o regular.
-G *archivo*	*archivo* existe y es propiedad del mismo grupo del usuario.
-h *archivo*	*archivo* existe y es un enlace simbólico.
-L *archivo*	*archivo* existe y es un enlace simbólico. Es igual que **-h**.
-O *archivo*	*archivo* existe y es propiedad del usuario.
-r *archivo*	*archivo* existe y el usuario tiene permiso de lectura sobre él.
-s *archivo*	*archivo* existe y es no vacío.
-w *archivo*	*archivo* existe y el usuario tiene permiso de escritura sobre él.
-x *archivo*	*archivo* existe y el usuario tiene permiso de ejecución sobre él, o es un directorio y el usuario tiene permiso de búsqueda en él.
archivo1 **-nt** *archivo2*	*archivo1* es más reciente que *archivo2*, según la fecha de modificación, o si *archivo1* existe y *archivo2* no.
archivo1 **-ot** *archivo2*	*archivo1* es más antiguo que *archivo2*, según la fecha de modificación, o si *archivo2* existe y *archivo1* no.
archivo1 **-ef** *archivo2*	*archivo1* es un enlace duro al *archivo2*, es decir, si ambos se refieren a los mismos números de dispositivo e *inode*.

Tabla 4.4. Operadores de consulta de archivos.

La orden **test** *expresión* es equivalente a la orden **[** *expresión* **]**, donde los huecos en blanco entre los corchetes y *expresión* son necesarios.

Ejemplo: A continuación se muestran algunos casos de evaluación de expresiones sobre archivos utilizando la orden `test` y corchetes:

```
$ test -d /bin      # comprueba si /bin es un directorio
$ echo $?           # nos muestra el estado de la última orden ejecutada, aunque
0                   # usado después de test o [ ] da 0 si la evaluación era verdadera
$ [ -w /bin ]       # comprueba si tenemos permiso de escritura en /bin
$ echo $?           # usado después de test o [ ] da 1 si la evaluación era falsa
1
$ test -f /bin/cat      # comprueba si el archivo /bin/cat existe y es plano
$ echo $?
0
$ [ /bin/cat -nt /bin/zz ]    # comprueba si /bin/cat es más reciente que /bin/zz
$ echo $?
0                             # la evaluación devuelve 0 porque /bin/zz no existe
```

Ejemplo: Vemos algunas características de archivos existentes en el directorio `/bin` y asignamos el resultado a una variable:

```
$ cd /bin
$ ls -l cat
-rwxr-xrx 1 root root 38524 2010-06-11 09:10 cat

$ xacceso=`test -x cat && echo "true" || echo "false"`    # se pueden omitir las "
$ echo $xacceso
true                  # indica que sí tenemos permiso de ejecución sobre cat

$ wacceso=`test -w cat && echo "true" || echo "false"`    # se pueden omitir las "
$ echo $wacceso
false                 # indica que no tenemos permiso de escritura en cat
```

Además, es posible varias expresiones de la orden test mediante empleando los operadores **`!`**, **`-a`** y **`–o`**, not, and y or respectivamente.

Ejemplo: Combinación de expresiones de la orden test

```
# Si el archivo "ejemplo" no tiene permisos de escritura
$ ls –la ejemplo
-rw-r--r--  1 usuario usuario 256 29 nov  2016 ejemplo

$ ejemploNOT=`! test –x ejemplo && echo true || echo false`
$ echo $ejemploNOT
true

# Si es propiedad del usuario que ejecuta la orden y no está vacío
$ ejemploAND=`test –O ejemplo –a –s ejemplo && echo true || echo false`
$ echo $ejemploAND
true

# Si tiene permisos de lectura o de escritura
$ ejemploOR=`test –r ejemplo –o –w ejemplo && echo true || echo false`
$ echo $ejemploOR
true
```

Ejercicio 4.10: Usando **`test`**, construya un guion que admita como parámetro un nombre de archivo y realice las siguientes acciones: asignar a una variable el resultado de comprobar si el archivo dado como parámetro es plano y tiene permiso de ejecución sobre él; asignar a otra variable el resultado de comprobar si el archivo es un enlace simbólico; mostrar el valor de las dos variables anteriores con un mensaje que aclare su significado. Pruebe el guion ejecutándolo con `/bin/cat` y también con `/bin/rnano`.

Ejercicio 4.11: Ejecute **`help test`** y anote qué otros operadores se pueden utilizar con la orden **`test`** y para qué sirven. Ponga un ejemplo de uso de la orden **`test`** comparando dos expresiones aritméticas y otro comparando dos cadenas de caracteres.

4.2.5 Orden if / else

La orden **`if/else`** ejecuta una lista de declaraciones dependiendo de si se cumple o no cierta condición, y se podrá utilizar tanto en la programación de guiones, como en expresiones más simples.

La sintaxis de la orden condicional **`if`** es:

```
if condición;
then
      declaraciones
[elif condición;
      then declaraciones ]...
[else
      declaraciones ]
fi
```

La principal diferencia de este condicional respecto a otros lenguajes es que cada *condición* representa realmente una lista de declaraciones, con órdenes, y no una simple expresión booleana. De esta forma, como las órdenes terminan con un estado de finalización, *condición* se considera *true* si su estado de finalización (`status`) es 0, y *false* en caso contrario (estado de finalización igual a 1). Al igual que en otros lenguajes, en cualquiera de las *declaraciones* puede haber otra orden **`if`**, lo que daría lugar a un anidamiento.

El funcionamiento de la orden **`if`** es el siguiente: se comienza haciendo la ejecución de la lista de órdenes contenidas en la primera *condición*; si su estado de salida es 0, entonces se ejecuta la lista de *declaraciones* que sigue a la palabra **`then`** y se termina la ejecución del **`if`**; si el estado de salida fuese 1, se comprueba si hay un bloque que comience por **`elif`**. En caso de haber varios bloques **`elif`**, se evalúa la *condición* del primero de ellos de forma que si su estado de salida es 0, se hace la parte **`then`** correspondiente y termina el **`if`**, pero si su estado de salida es 1, se continúa comprobando de manera análoga el siguiente bloque **`elif`**, si es que existe. Si el estado de salida de todas las condiciones existentes es 1, se comprueba si hay un bloque **`else`**, en cuyo caso se ejecutarían las *declaraciones* asociadas a él, y termina el **`if`**.

En el ejemplo siguiente se utiliza la orden **`if`** para tener una estructura similar a la que se había planteado anteriormente con **`test`** usando "**`&&`**" y "**`||`**":

```
$ cd /bin
$ ls -l cat
-rwxr-xrx 1 root root 38524 2010-06-11 09:10 cat

$ xacceso=`if test -x cat; then echo "true"; else echo "false"; fi`
$ echo $xacceso
true                    # indica que sí tenemos permiso de ejecución sobre cat

$ wacceso=`if test -w cat; then echo "true"; else echo "false"; fi`
$ echo $wacceso
false                   # indica que no tenemos permiso de escritura en cat
```

Como se puede apreciar, la condición de la orden **`if`** puede expresarse utilizando la orden **`test`** para hacer una comprobación. De forma análoga se puede utilizar **`[ ... ];`** si se usa "`if [ expresión ];`", *expresión* puede ser una expresión booleana y puede contener órdenes, siendo necesarios los huecos en blanco entre los corchetes y *expresión*.

Ejercicio 4.12: Responda a los siguientes apartados:

a) Razone qué hace la siguiente orden:
`if test -f ./sesion5.pdf ; then printf "El archivo ./sesion5.pdf existe\n"; fi`

b) Añada los cambios necesarios en la orden anterior para que también muestre un mensaje de aviso en caso de no existir el archivo. (Recuerde que, para escribir de forma legible una orden que ocupe más de una línea, puede utilizar el carácter "`\`" como final de cada línea que no sea la última.)

c) Sobre la solución anterior, añada un bloque **`elif`** para que, cuando no exista el archivo `./sesion5.pdf`, compruebe si el archivo `/bin` es un directorio. Ponga los mensajes adecuados para conocer el resultado en cada caso posible.

d) Usando como base la solución del apartado anterior, construya un guion que sea capaz de hacer lo mismo pero admitiendo como parámetros la ruta relativa del primer archivo a buscar y la ruta absoluta del segundo. Pruébelo con los dos archivos del apartado anterior.

Ejercicio 4.13: Construya un guion que admita como argumento el nombre de un archivo o directorio y que permita saber si somos el propietario del archivo y si tenemos permiso de lectura sobre él.

4.2.5.1 Comparaciones aritméticas

La comparación entre valores aritméticos se puede realizar empleando los operadores citados en la tabla 4.3. A la hora de efectuar comparaciones con valores numéricos es posible tratar dichos valores como cadenas de caracteres o como su número. A continuación mostramos ejemplos para ambos casos.

El ejemplo que viene a continuación plantea algunos casos de comparaciones aritméticas utilizando corchetes y tratando la comparativa como caracteres (de ahí que el valor 34 esté entre comillas):

```
$ valor=34
$ if [ $valor == "34" ]; then echo sí; else echo no; fi # los huecos en blanco a los
sí                                                      # lados de los operadores
$ if [ $valor -eq "34" ]; then echo sí; else echo no; fi# relacionales son
sí                                                      # necesarios
$ if [ $valor == "40" ]; then echo sí; else echo no; fi
no
```

Si lo que se desea es comparar la igualdad con el valor numérico, en ese caso debemos usar el operador **=**.

```
$ valor=34
$ if [ $valor = 34 ]; then echo sí; else echo no; fi
sí
$ if [ $valor = 40 ]; then echo sí; else echo no; fi
no
```

También podemos realizar comparaciones aritméticas utilizando el doble paréntesis. En el siguiente ejemplo se muestra cómo hacerlo.

```
$ var1=234
$ var2=456
$ if (( $var1 <= $var2 )); then echo sí; else echo no; fi
sí
```

```
$ if (( $var1 != $var2 )); then echo sí; else echo no; fi
sí
```

```
$ if (( $var1 == 234 )); then echo sí; else echo no; fi
sí
```

Cuando se emplea el doble paréntesis, no es posible utilizar los operadores **-eq**, **-ne**, **-le**, **-lt**, **-ge** y **-gt**. En su lugar se utilizarán los operadores equivalentes mostrados en la tabla 4.3. Además, la última comparación obliga a usar el operador de igualdad **==**. Si se emplea el **=** responderá con un mensaje de error.

```
$ if (( $var1 = 234 )); then echo sí; else echo no; fi no
-bash: ((: 234 = 234 : se intentó asignar a algo que no es una variable (el elemento
de error es "= 234 ")
no
```

4.3.5.2 Comparaciones entre cadenas de caracteres

En bash, los operadores **==** y **!=**, también pueden utilizarse para comparar si dos cadenas de caracteres `A` y `B` coinciden o no, respectivamente; además, en la versión 2.0 y las posteriores, los operadores **<** y **>** permiten comparar si una cadena de caracteres `A` se clasifica antes o después de otra cadena `B`, respectivamente, siguiendo el orden lexicográfico. Además, si se deja un espacio en blanco antes y después del operador **=**, también se realiza una comparación en lugar de una asignación si se hace sin los pertinentes espacios en blanco.

```
$ valor=”hola”
$ if [ $valor = "hola" ]; then echo sí; else echo no; fi
sí
$ if [ $valor == "hola" ]; then echo sí; else echo no; fi
sí
$ if [ $valor=="adios" ]; then echo sí; else echo no; fi
si
$ if [ $valor == "adios" ]; then echo sí; else echo no; fi
no

$ if [ $valor != "adiós" ]; then echo sí; else echo no; fi
sí
```

Cuando la variable de caracteres que se desea comparar pueda contener algún espacio en blanco, es obligatorio poner la variable entre comillas para realizar correctamente la comparación (cuando la variable contiene un valor numérico o una palabra, no es necesario representarla entre comillas). En el siguiente ejemplo se ilustra tal necesidad.

```
$ valor=”hola amigos”
$ if [ $valor == "hola amigos" ]; then echo sí; else echo no; fi
-bash: [: demasiados argumentos
no
```

```
$ if [ “$valor” == "hola amigos" ]; then echo sí; else echo no; fi
sí
```

4.3.5.3 Comparaciones usando órdenes

En los ejemplos siguientes la `condición` del **`if`** es una orden:

```
$ valor=6
$ if [ valor=3 ]; then echo sí; else echo no; fi
sí                    # se hace internamente la orden que hay entre corchetes y no
$ echo $valor         # da error, pero la supuesta asignación en la condición del
6                     # if no tiene efecto sobre la variable que se estaba usando

$ if ls > salida; then echo sí; else echo no; fi
sí                     # además, el if hace la orden ls sobre el archivo salida
$ cat salida           # vemos que la orden ls anterior ha volcado su resultado en
                       # este archivo, poniendo cada nombre en una línea distinta
                       # e incluyendo también el nombre “salida”
$ valor=5
$ if valor=3 && ls ; then echo sí; else echo no; fi
                # muestra el resultado de ls y otra línea con sí;
                # además, hace la asignación correctamente, tal como podemos ver

$ echo $valor
```

```
3               # el if ha cambiado el contenido de la variable valor
$ if rm salida; then echo sí; else echo no; fi 2> sal
                # en caso de que el archivo salida exista antes del if, se borra y
                # escribe una línea en pantalla poniendo sí;
                # en caso de que ese archivo no exista, escribe en pantalla una
                # línea con no y pone un mensaje de error en el archivo sal
```

Ejercicio 4.14: Escriba un guion que calcule si el número de días que faltan hasta fin de año es múltiplo de cinco o no, y que comunique el resultado de la evaluación. Modifique el guion anterior para que admita la opción `-h` de manera que, al ejecutarlo con esa opción, muestre información de para qué sirve el guion y cómo debe ejecutarse.

El siguiente guion de ejemplo se puede utilizar para borrar el archivo `temporal` que se le dé como argumento. Si **`rm`** devuelve 0, se muestra el mensaje de confirmación del borrado; en caso contrario, se muestra el código de error. Como se puede apreciar, hemos utilizado la variable `$LINENO` que indica la línea actualmente en ejecución dentro del guion.

```
#!/bin/bash
declare -rx SCRIPT=${0##*/}  # donde SCRIPT contiene sólo el nombre del guión
                             # ${var##Patron} actúa eliminando de $var aquella parte
                             # que cumpla de $Patron desde el principio de $var
                             # En este caso: elimina todo lo que precede al
                             # último slash "/".

if rm ${1} ; then
      printf "%s\n" "$SCRIPT: archivo temporal borrado"
else
      STATUS=177
      printf "%s – código de finalizacion %d\n" \
      "$SCRIPT:$LINENO no es posible borrar archivo" $STATUS
fi 2> /dev/null
```

En la siguiente referencia se puede encontrar información adicional sobre la sustitución de parámetros y algunos ejemplos: `http://tldp.org/LDP/abs/html/parameter-substitution.html`

Ejercicio 4.15: ¿Qué pasa en el ejemplo anterior si eliminamos la redirección de la orden **`if`**?

De igual modo, es posible usar la orden **`test`** dentro de la comparativa de una orden **`if`**. En el siguiente ejemplo se ilustra mejor tal utilización.

```
$ valor=123
$ if test $valor -eq 123; then echo sí; else echo no; fi
sí
$ if test $valor = 123; then echo sí; else echo no; fi
sí
```

```
$ if test $valor = "123"; then echo sí; else echo no; fi
sí
```

```
$ if test $valor == 123; then echo sí; else echo no; fi
sí
```

```
$ if test "$valor" == 123; then echo sí; else echo no; fi
sí
$ if test "$valor" = 123; then echo sí; else echo no; fi
sí
```

```
$ if test $valor = 12333; then echo sí; else echo no; fi
no
$ if test $valor = "12333"; then echo sí; else echo no; fi
no
```

4.3 Expresiones regulares

Una *expresión regular* es un patrón que describe un conjunto de cadenas y que se puede utilizar para búsquedas dentro de una cadena o un archivo. Un usuario avanzado o un administrador del sistema que desee obtener la máxima potencia de ciertas órdenes debe conocer y manejar expresiones regulares. Las expresiones regulares se construyen de forma similar a las expresiones aritméticas, utilizando diversos operadores para combinar expresiones más sencillas. Las piezas fundamentales para la construcción de expresiones regulares son las que representan a un carácter simple. La mayoría de los caracteres, incluyendo las letras y los dígitos, se consideran expresiones regulares que se representan a sí mismos. Para comparar con el carácter asterisco (*****), éste debe ir entre comillas simples (**'*'**).

En las expresiones regulares se puede utilizar una barra inclinada invertida (\), denominada a veces como *barra de escape*, para modificar la forma en la que se interpretará el carácter que le siga. Cuando los metacaracteres "?", "+", "{", "}", "|", "(" y ")" aparecen en una expresión regular no tienen un significado especial, salvo que vayan precedidos de una barra de escape; si se utilizan anteponiendo esa barra, es decir "\?", "\+", "\{", "\}", "\|", "\(" o "\)", su significado será el que corresponda según la Tabla 4.5.

Para practicar con expresiones regulares puede ser útil la página web http://rubular.com , que dispone de una interfaz en la que se puede escribir un texto y probar diferentes expresiones regulares aplicables sobre el texto y, de manera interactiva, el editor mostrará qué se va seleccionando en función de la expresión regular escrita.

4.3.1 Patrones en expresiones regulares

La Tabla 4.5 muestra patrones simples que pueden utilizarse en expresiones regulares. Una expresión regular estará formada por uno o varios de estos patrones.

Dos expresiones regulares también pueden concatenarse; la expresión que se obtiene representa a cualquier cadena formada por la concatenación de las dos subcadenas dadas por las subexpresiones concatenadas. Esto no debe confundirse con el uso del operador OR mencionado en la Tabla 4.5.

Por defecto, las reglas de precedencia indican que primero se trata la repetición, luego la concatenación y después la alternación, aunque el uso de paréntesis permite considerar subexpresiones y cambiar esas reglas.

4.3.2 Expresiones regulares con órdenes de búsqueda

Las órdenes de búsqueda **`find`**, **`grep`** y **`egrep`** dadas en la práctica anterior, adquieren mayor relevancia al hacer uso de expresiones regulares en ellas, cosa que se verá con algunos casos. Es conveniente recordar las múltiples y útiles opciones de estas órdenes, además de una diferencia fundamental entre ellas: **`find`** sirve para buscar a nivel de características generales de los archivos, como, por ejemplo, nombre o condiciones de acceso, pero sin entrar en su contenido; por el contrario, **`grep`** y **`egrep`** examinan la cadena que se le dé mediante la entrada estándar o el contenido de los archivos que se le pongan como argumento.

Dado que la shell bash intenta interpretar algunos caracteres antes de pasárselos a las órdenes de búsqueda, es especialmente importante tener en cuenta lo mencionado anteriormente sobre la barra de escape en el uso de expresiones regulares.

A continuación proponemos algunos ejemplos y ejercicios que debe probar para aprender a realizar búsquedas con patrones más o menos complejos.

Ejemplo: Buscar en el directorio `/bin/usr` los archivos cuyo nombre comience con las letras `a` o `z` y acabe con la letra `m`:

```
$ find /usr/bin -name "[az]*m"
```

Patrón	Representa
****	la barra de escape; si en un patrón se quiere hacer referencia a este mismo carácter, debe ir precedido por él mismo y ambos entre comillas simples.
.	cualquier carácter en la posición en la que se encuentre el punto cuando se usa en un patrón con otras cosas; si se usa solo, representa a cualquier cadena; si se quiere buscar un punto como parte de un patrón, debe utilizarse `\.` entre comillas simples o dobles.
( )	un grupo; los caracteres que se pongan entre los paréntesis serán considerados conjuntamente como si fuesen un único carácter. (Hay que usar `\`)
?	que el carácter o grupo al que sigue puede aparecer una vez o no aparecer ninguna vez. (Hay que usar `\`)
*****	que el carácter o grupo al que sigue puede no aparecer o aparecer varias veces seguidas. (No hay que usar `\`)
+	que el carácter o grupo previo debe aparecer una o más veces seguidas.
{n}	que el carácter o grupo previo debe aparecer exactamente *n* veces. (Hay que usar `\`)
{n,}	que el carácter o grupo previo debe aparecer *n* veces o más seguidas. (Hay que usar `\`)
{n,m}	que el carácter o grupo previo debe aparecer de *n* a *m* veces seguidas; al menos *n* veces, pero no más de *m* veces. (Hay que usar `\`)
[ ]	una lista de caracteres que se tratan uno a uno como caracteres simples; si el primer carácter de la lista es "^", entonces representa a cualquier carácter que no esté en esa lista.
-	un rango de caracteres cuando el guion no es el primero o el último en una lista; si el guion aparece el primero o el último de la lista, entonces se trata como él mismo, no como rango; en los rangos de caracteres, el orden es el alfabético, pero intercalando minúsculas y mayúsculas – es decir: aAbB...-; en los rangos de dígitos el orden es 012... También es posible describir rangos parciales omitiendo el inicio o el final del rango (por ejemplo [m-] representa el rango que va desde la "m" hasta la "z").
^	indica el inicio de una línea; como se ha dicho anteriormente, cuando se usa al comienzo de una lista entre corchetes, representa a los caracteres que no están en esa lista. Situando a continuación de `^` un carácter, filtrará todas aquellas líneas que comiencen por ese carácter.
$	indica el final de una línea. Situando un carácter antes del `$`, filtrará todas aquellas líneas que terminen por ese carácter.
\b	el final de una palabra. (Debe utilizarse entre comillas simples o dobles)
\B	que no está al final de una palabra. (Debe utilizarse entre comillas simples o dobles)
\<	el comienzo de una palabra. (Debe utilizarse entre comillas simples o dobles)
\>	el final de una palabra. (Debe utilizarse entre comillas simples o dobles)
|	el operador OR para unir dos expresiones regulares, de forma que la expresión regular resultante representa a cualquier cadena que coincida con al menos una de las dos subexpresiones. (La expresión global debe ir entre comillas simples o dobles; además, cuando se usa con `grep`, esta orden debe ir acompañada de la opción `-E`)

Tabla 4.5. Operadores para expresiones regulares.

Ejemplo: Buscar en el directorio `/etc` los archivos de configuración que contengan la palabra "`dev`":

```
$ find /etc -name "*dev*.conf"
```

Seguidamente se utiliza la orden **`find`** con la opción **`-regex`** para especificar que se buscan nombres de archivos que satisfagan la expresión regular que se pone a continuación de esta opción (v.g., "regex" significa "regular

expression"). Si no se usa esta opción se interpretarán los símbolos como metacaracteres y no como parte de una expresión regular.

El ejemplo siguiente se ha resuelto de dos formas equivalentes y sirve para localizar en el directorio `/usr/bin` los nombres de archivos que contengan las letras `cd` o `zip`:

```
$ find  /usr/bin  -regex  '.*\(cd\|zip\).*'

$ find  /usr/bin  -regex  '.*cd.*' -o -regex  '.*zip.*'
```

Ejemplo: Buscar en el archivo `/etc/group` las líneas que contengan un carácter `k`, o `y` o `z`:

```
$ grep [kyz] /etc/group
```

El ejemplo anterior también se puede resolver de otro modo que parece más complicado, pero que aumenta la capacidad de programación para construir guiones. Haciendo uso del mecanismo de cauces que nos ofrece Linux, el archivo cuyo contenido se examina con la orden **`grep`** se puede sustituir por la salida de una orden **`cat`** sobre ese mismo archivo:

```
$ cat  /etc/group  |  grep [kyz]
```

De manera análoga, utilizando el mecanismo de cauces se puede conseguir que la salida de otras órdenes se trate como si fuese un archivo cuyo contenido se desea examinar.

El siguiente ejemplo permite añadir a la variable `$PATH` un directorio que se pase como primer argumento del guion; si se da la opción `after` como segundo argumento del guion, el directorio se añadirá al final de la variable `$PATH`, en cualquier otro caso el directorio se añadirá al principio de `$PATH`:

```
#!/bin/bash
# Uso: pathmas directorio [after]
if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
          if [ "$2" = "after" ] ; then
              PATH=$PATH:$1
           else
              PATH=$1:$PATH
          fi
else
    echo "$1 ya está en el path"
fi
```

Ejercicio 4.16: Haciendo uso del mecanismo de cauces y de la orden **`echo`**, construya un guion que admita un argumento y que informe si el argumento dado es una única letra, en mayúsculas o en minúsculas, o es algo distinto de una única letra.

Ejercicio 4.17: Haciendo uso de expresiones regulares, escriba una orden que permita buscar en el árbol de directorios los nombres de los archivos que contengan al menos un dígito y la letra `e`. ¿Cómo sería la orden si quisiéramos obtener los nombres de los archivos que tengan la letra `e` y no contengan ni el `0` ni el `1`?

Ejercicio 4.18: Utilizando la orden **`grep`**, exprese una forma alternativa de realizar lo mismo que con **`wc -l`**.

4.3.3 Ejemplos de utilización de expresiones regulares

Para finalizar esta práctica y afianzarse en el uso de expresiones regulares utilice el archivo `p4_Complementos.txt` que hay en la plataforma tutor. En él encontrará instrucciones para crear un archivo de prueba y ejecutar algunos ejemplos de filtrado del mismo a través de la orden **`grep`** acompañada de la descripción del patrón de búsqueda por medio de expresiones regulares.

Además de lo anterior, están disponibles otra serie de ejercicios para afianzar aún más los conocimientos acerca de este apartado.

Práctica 5: Programación del shell

5.1 Objetivos principales

- Aprender a utilizar las órdenes de control de flujo para alterar la ejecución secuencial de órdenes y cómo podemos pasar información a un guion desde el teclado.
- Saber definir y usar funciones dentro del bash shell.
- Ampliar el repertorio de órdenes de Unix visto hasta el momento.

Además, se verán las siguientes órdenes:

Órdenes Linux			
:	for	while	until
case	select	seq	read
break	true	continue	return
tar	gzip	cut	

Tabla 5.1. Órdenes de la práctica.

En esta práctica, veremos cómo construir guiones más complejos, introduciendo nuevas órdenes, que si bien pueden ser usadas de modo interactivo (como vimos que ocurría con `if` en la Práctica 4), muestran su gran potencia en programas shell. Las órdenes de control de flujo que veremos son:

- **`for`** ejecuta una lista de declaraciones un número fijo de veces.
- **`while`** ejecuta una lista de declaraciones cierto número de veces mientras cierta condición se cumple.
- **`until`** ejecuta una lista de declaraciones repetidamente hasta que se cumpla cierta condición.
- **`case`** ejecuta una de varias listas de declaraciones dependiendo del valor de una variable.

5.2 Lectura del teclado

En nuestros guiones podemos leer desde el teclado usando la orden **`read`**, que detiene la ejecución del guion y espera a que el usuario teclee algo de forma que el texto tecleado es asignado a la(s) variable(s) que acompañan a la orden. Su sintaxis es:

```
read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-p prompt] [-t timeout] [-u fd] [name …]
```

Podéis consultar las opciones con `help read`. Por ejemplo, podemos leer el nombre de un archivo de la forma:

```
#!/bin/bash

printf "Introduzca el nombre del archivo:"
read ARCHIVO_INPUT
printf "\nEl nombre del archivo es: %s" $ARCHIVO_INPUT
```

O bien se puede hacer de esta forma:

```
#!/bin/bash

read -p "Introduzca el nombre del archivo:" ARCHIVO_INPUT
printf "\nEl nombre del archivo es: %s" $ARCHIVO_INPUT
```

En ambos casos, la variable `$ARCHIVO_INPUT` contiene los caracteres tecleados por el usuario.

5.3 Orden for

El bucle for permite repetir cierto número de veces las declaraciones especificadas. Su sintaxis es:

```
for nombre [in lista]
do
      declaraciones que pueden usar $nombre
done
```

Podemos ver de qué tipo son todos los archivos de nuestro directorio de trabajo con el siguiente guion:

```
#!/bin/bash

for archivo in $(ls)
do
   file $archivo
done
```

En el siguiente ejemplo mostramos los 5 primeros números impares mediante la orden **`seq`**, que genera una secuencia de números desde el valor indicado como primer argumento (éste será el valor inicial), hasta el valor final dado como tercer argumento, tomando como incremento el segundo.

```
#!/bin/bash
for NUM in `seq 0 1 4`;
do
 let "NUM=$NUM * 2 + 1"
 printf "Número impar %d\n" $NUM
done
```

Podemos construir un bucle clásico de C `(())`, como muestra el ejemplo siguiente:

```
#!/bin/bash
# forloop.sh: Cuenta desde 1 a 9
for (( CONTADOR=1; CONTADOR<10; CONTADOR++ )) ; do
  printf "Contador vale ahora %d\n" $CONTADOR
done
```

En este bucle se declara e inicializa la variable `CONTADOR=1` y el bucle finaliza cuando la variable vale `10 (CONTADOR<10)`. En cada iteración, la variable se incrementa en `1 (CONTADOR++)`. Es posible que la condición tenga varias variables.

Ejercicio 5.1. Escriba un guion que acepte dos argumentos. El primero será el nombre de un directorio y el segundo será un valor entero. El funcionamiento del guion será el siguiente: deberán anotarse en un archivo denominado `archivosSizN.txt` aquellos archivos del directorio dado como argumento y que cumplan la condición de tener un tamaño menor al valor aportado en el segundo argumento. Se deben tener en cuenta las comprobaciones sobre los argumentos, es decir, debe haber dos argumentos, el primero deberá ser un directorio existente y el segundo un valor entero.

Ejercicio 5.2. Escriba un guion que acepte el nombre de un directorio como argumento y muestre como resultado el nombre de todos y cada uno de los archivos del mismo y una leyenda que diga "Directorio", "Enlace" o "Archivo regular", según corresponda. Incluya la comprobación necesaria sobre el argumento, es decir, determine si el nombre aportado se trata de un directorio existente.

5.4 Orden case

La orden **case** compara una variable con una serie de valores o patrones. Si el valor o el patrón coinciden, se ejecutan las declaraciones correspondientes. La sintaxis para esta orden es:

```
case expresión in
      patron1 )
            declaraciones;;
      patron2 )
            declaraciones;;
      …
esac
```

Como ejemplo, vamos a construir un guion que permite al usuario elegir entre dos opciones. Si elegimos la primera, borramos un archivo; si es la segunda, hacemos una copia de seguridad de él. Se pedirá al usuario que introduzca lo necesario de forma interactiva mediante el teclado. Debe crear el archivo `temporal` dentro del directorio `tmp`.

```
#!/bin/bash
printf "%s -> " "1 = borrar, 2 = copiar. Elija una opción"
TEMPFILE=/tmp/temporal
read REPLY
case "$REPLY" in
      1) rm "$TEMPFILE" ;;
      2) mv "$TEMPFILE" "$TEMPFILE.old" ;;
      *) printf "%s\n" "$REPLY no es una de las opciones" ;;
esac
```

En la línea 8 del ejemplo anterior hemos puesto el patrón asterisco `(*)` que representa a cualquier otro caso, de esta forma, si no se ha dado una de las respuestas anteriores se ejecutarán las declaraciones de esta parte.

Es posible añadir distintos patrones dentro de una misma etiqueta de la orden `case`. Para ello, se usa el símbolo `|` como separador (ver el último ejemplo del apartado 5, pág. 68). Observe que en esta ocasión, no se interpreta como operador de cauce y sí como delimitador de diferentes opciones.

Ejercicio 5.3. Escriba un guion en el que, a partir de la pulsación de una tecla, detecte la zona del teclado donde se encuentre. Las zonas vendrán determinadas por las filas. La fila de los números `1, 2, 3, 4, ...` será la fila 1, las teclas donde se encuentra la `Q, W, E, R, T, Y,...` serán de la fila 2, las teclas de la `A, S, D, F, ...` serán de la fila 3 y las teclas de la `Z, X, C, V, ...` serán de la fila 4. La captura de la tecla se realizará mediante la orden **read**.

Ejercicio 5.4. Escriba un guion que acepte como argumento un parámetro en el que el usuario indica el mes que quiere ver, ya sea en formato numérico o usando las tres primeras letras del nombre del mes, y muestre el nombre completo del mes introducido. Si el número no está comprendido entre 1 y 12 o las letras no son significativas del nombre de un mes, el guion deberá mostrar el correspondiente mensaje de error.

5.5 Órdenes while y until

Las órdenes **while** y **until** permiten repetir una serie de declaraciones. La orden **while** ejecuta las declaraciones comprendidas entre **do** y **done** mientras que la expresión sea *true* (`expresión` puede ser una condición o una expresión de control). Si `expresión` falla en el primer intento, nunca se ejecutan las declaraciones del cuerpo de **while**. La sintaxis para **while** es:

```
while expresión;
do
      declaraciones …
done
```

Es importante que comprenda que este bucle encontrará falsa su condición cuando demos <control-d> (que siempre tendrá el significado de fin de archivo). Observe la potencialidad de este lenguaje en el que con pocas instrucciones estamos consiguiendo una funcionalidad grande. Una vez que se almacene en la variable `$ARCHIVO` un valor, en la línea 5 la orden **`test`** chequea si existe o no dicho archivo cuyo nombre ha introducido el usuario por teclado.

```
#!/bin/bash
printf "%s\n" "Introduzca los nombres de archivos o <control-d> para finalizar"
while read -p "Archivo ?" ARCHIVO ;
do
   if test -f "$ARCHIVO" ;
   then
      printf "%s\n" "El archivo existe"
   else
      printf "%s\n" "El archivo NO existe"
   fi
done
```

También podemos combinar la orden `while` con `read` para leer de un archivo como muestra el ejemplo siguiente:

```
#!/bin/bash
# Leer datos de un archivo
count=1
cat /etc/passwd | while read linea ;
do
   echo "Linea $count: $linea"
   count=$[ $count + 1 ]
done
echo "Fin de procesamiento del archivo"
```

Otra forma de hacer lo mismo utilizando redirecciones, sería:

```
#!/bin/bash
# Leer datos de un archivo
count=1
while read line ;
do
   echo "Line $count: $line"
   count=$[ $count + 1 ]
done < test
echo "Finished processing the file"
```

Podemos crear un bucle infinito utilizando la orden **`true`**. Dado que **`true`** siempre tiene éxito, el bucle se ejecuta indefinidamente:

```
#!/bin/bash
printf "%s\n" "Introduzca los nombres de archivos o teclea FIN"
while true ;
do
   read -p "Archivo ?" ARCHIVO
   if [ "$ARCHIVO" = "FIN" ] ; then
      break
   elif test -f "$ARCHIVO" ; then
       printf "%s\n" "El archivo existe"
   else
       printf "%s\n" "El archivo NO existe"
```

```
   fi
done
```

Hemos utilizado la orden **break** para finalizar el bucle si se escribe `FIN`. Cuando el shell encuentra esta orden, sale del bucle (orden **while**) y continúa ejecutando la orden siguiente. La orden **break** puede venir seguida de un número que indica cuántos bucles anidados romper, por ejemplo, `break 2`.

Otra forma de describir un bucle infinito sería utilizando una variable cuya asignación inicial fuese el valor `true` y, cuando se deseara terminar, asignarle el valor `false`. Basándonos en el ejemplo anterior, la estructura quedaría así:

```
#!/bin/bash
printf "%s\n" "Introduzca los nombres de archivos o teclea FIN"
sigue=true
while $sigue ;
do
   read -p "Archivo ?" ARCHIVO
   if [ "$ARCHIVO" = "FIN" ] ; then
      sigue=false
   elif test -f "$ARCHIVO" ; then
       printf "%s\n" "El archivo existe"
   else
       printf "%s\n" "El archivo NO existe"
   fi
done
```

Como se puede apreciar, hemos sustituido la orden `break` para salir del bucle por una asignación de la variable que controla la iteración en el bucle.

Otra orden ligada a los bucles es la orden **continue**, que permite iniciar una nueva iteración del bucle saltando las declaraciones que haya entre ella y el final del mismo. Por ejemplo, podemos generar hasta nueve números aleatorios siempre y cuando sean mayores de 20000. Para ello, utilizamos la variable `$RANDOM` que devuelve un entero diferente entre 0 y 32676 cada vez que la consultamos.

```
#!/bin/bash
for n in {1..9} ## Ver uso de llaves en sesiones anteriores
do
   x=$RANDOM
   [ $x -le 20000 ] && continue
   echo "n=$n x=$x"
done
```

La orden **until** es similar a **while** excepto que se repite el cuerpo hasta que la condición sea cierta. En este caso, la condición se comprueba antes de cada iteración, incluso antes de la primera iteración. La sintaxis para **until** es:

```
until expresión;
do
      declaraciones …
done
```

Ahora, vamos a ver un ejemplo donde expresión es una condición en lugar de una orden, como ocurría en los ejemplos anteriores. El ejemplo genera los 10 primeros números:

```
#!/bin/bash
n=1
until [ $n -gt 10 ]
do
   echo "$n"
```

```
   n=$(( $n + 1 ))
done
```

Algo bastante frecuente es la construcción de menús, a continuación vemos un ejemplo. En la línea 1 aparece como expresión del bucle **while** la orden **:** que es equivalente a "no operación". Además, una vez elegida la opción dentro de la orden **case**, hacemos que, mediante la orden **sleep 2**, espere durante 2 segundos hasta que vuelva a solicitarnos una de las opciones del menú.

```
#!/bin/bash
while :
do
   printf "\n\nMenu de configuración:\n"
   printf "\tInstalar ...  [Y, y]\n"
   printf "\tNo instalar [N, n]\n"
   printf "\tPara finalizar pulse 'q'\n"

   read -n1 -p "Opción: " OPCION
   case $OPCION in
       Y | y ) printf "\nHas seleccionado %s \n" $OPCION;;
       N | n ) printf "\nHas seleccionado %s \n" $OPCION;;
       q ) printf "\n"
           break;;
       * ) printf "\n Selección inválida: %s \n";;
   esac
   sleep 2  # duerme durante 2 segundos
done
```

Ejercicio 5.5. Escriba un guion que solicite un número hasta que su valor esté comprendido entre 1 y 10. Deberá usar la orden **while** y, para la captura del número, la orden **read**.

5.6 Funciones

Una de las características de Bash es que nos permite definir funciones. Éstas, a diferencia de los guiones, se ejecutan dentro de la memoria del proceso bash que las utiliza, por lo que su invocación es más rápida que invocar a un guion u orden de Unix. Definimos una función con la orden **function** y luego el nombre de la función o, del mismo modo, sin tener que escribir la palabra anterior pero añadiendo el paréntesis abierto y cerrado junto al nombre. Se puede apreciar su sintaxis a continuación:

```
function nombre_fn {
     declaraciones
}
```

```
nombre_fn() {
     declaraciones
}
```

De forma inversa, podemos borrar una función con **unset -f nombre_fn**. Podemos ver qué funciones tenemos definidas junto con su definición con **declare -f,** y solo su nombre con **declare -F**. ¿Qué ocurre si una función tiene el mismo nombre que un guion o una orden? El shell siempre utiliza el orden de preferencia que se cita a la hora de resolver un símbolo:

1. Alias
2. Palabra clave (como if, function, etc.)
3. Funciones
4. Órdenes empotradas
5. Guiones y programas ejecutables

Como norma de estilo, para evitar la confusión de nombres entre funciones y órdenes, podemos nombrar las funciones con un signo "_" delante del nombre (por ejemplo, `_mifuncion`). De todas formas, volviendo a la pregunta, y según la ordenación citada, una función con el mismo nombre que una orden empotrada se ejecutaría antes que el guion.

5.6.1 Variables locales en funciones. Parámetros

Para invocar una función dentro de un guion solamente debemos escribir su nombre seguido de los argumentos correspondientes, si los hubiera. En este sentido, las funciones utilizan sus propios parámetros posicionales.

Una función puede tener variables locales. Para ello, debemos declararlas dentro de la función con el modificador **`local`**. Por ejemplo:

```
#!/bin/bash
# file.sh: guion shell para mostrar el uso de funciones
# ------------------------------------------------------

# definimos la función _uso()
_uso()
{
   echo "Uso: $0 nombre_archivo"
   exit 1
}

# ------------------------------------------------------
# definimos la función _si_existe_file
# $f -> almacena los argumentos pasados al guion
_si_existe_file()
{
   local f="$1"
   [ -f "$f" ] && return 0 || return 1
}

# llamamos a _uso() si no se da el nombre de archivo
[ $# -eq 0 ] && _uso

# invocamos a _si_existe_file()
if ( _si_existe_file "$1" )
then
   echo "El archivo existe"
else
   echo "El archivo NO existe"
fi
```

Es posible utilizar los valores devueltos por una función. En el ejemplo siguiente, la función devuelve el doble de un número que se pide por teclado; en la línea 10 recogemos el valor devuelto por `_dbl` en la variable `$resultado`:

```
#!/bin/bash
# usamos echo para devolver un valor

function _dbl
{
   read -p "Introduzca un valor: " valor
   echo $[ $valor * 2 ]
}

resultado=`_dbl`
echo "El nuevo valor es $resultado"
```

Observe que, para ejecutar la función, se ha puesto el nombre de la misma entre comillas las interpretativas.

5.7 Ejemplos realistas de guiones

El objetivo de este apartado es mostrar algunos ejemplos de guiones completos para ilustrar todo su potencial.

5.7.1 Elimina directorios vacíos

El siguiente guion podrá ser llamado con argumentos o sin ellos. Si hay argumentos, deseamos que se borren los directorios vacíos que cuelguen de cada uno los argumentos indicados (que deben ser nombres de directorios existentes). Si no hay argumentos, deseamos que la operación se realice sobre el directorio actual. Observe la forma en que ésto se ha programado en el guion siguiente.

```
#!/bin/bash
# rmemptydir - remove empty directories
# Heiner Steven (heiner.steven@odn.de), 2000-07-17
#
# Category: File Utilities

[ $# -lt 1 ] && set -- .

find "$@" -depth -print  -type d |
while read dir
do
   [ `ls "$dir" | wc -l` -lt 1 ] || continue
   echo >&2 "$0: removing empty directory: $dir"
   rmdir "$dir" || exit $?
done
exit 0
```

En la línea 9 la orden **`find`** alude con "`$@`" a todos los parámetros posicionales ("`$@`" será sustituido por `$1`, `$2`, … hasta el final); pero si no se han puesto parámetros, ¿cómo conseguimos que únicamente con esta orden se aluda a `.`?

Vea lo que se hace en la línea 7: si el número de argumentos es <1 entonces deseamos que los parámetros posicionales sean "reemplazados" por ., y esto es justamente lo que hace **`set -- .`** (Consulte la ayuda de bash para ver la explicación completa y detallada de las órdenes empotradas (*shell builting commands*), en concreto `set`).

¿Cuál es la entrada estándar de la orden `read` que aparece en la línea 10? la salida de la orden `find`, por esa razón la variable `$dir`, en el cuerpo del bucle irá tomando los sucesivos valores que va proporcionando `find`, y el bucle terminará cuando acabe esta lista.

Una vez visto esto, el tratamiento para cada ruta encontrada por `find` es simple: en la línea 12 se chequea el número de hijos, si es menor que 1 no se ejecutará la orden `continue` y el guion seguirá su ejecución con la orden `echo`, pero si no es menor que 1, se ejecuta la orden `continue` terminando esa iteración del bucle sin hacer nada con este directorio y pasaría a tratar el siguiente si lo hubiera.

En la línea 14, si ha dado error la ejecución de la orden `rmdir`, entonces finalizaría devolviendo el mismo código de error que nos hubiera devuelto la citada orden. Si no hemos pasado por aquí, entonces la orden de la línea 16 devolverá el valor 0 indicativo de que no ha habido error. En ambos casos, se emplea la orden `exit` que abandona la ejecución de un guion. Dicha orden permite añadir un valor entero representativo de acuerdo a la dinámica de trabajo de las órdenes de la shell bash, es decir, con éxito devuelven el valor 0 y si hubo algún error, devolver el valor 1. De hecho, en el ejemplo anterior, se devolverá el valor de la ejecución de la última orden mediante el resultado de **`$?`**.

5.7.2 Muestra información del sistema en una página html

En el ejemplo siguiente se construye el archivo `status.html` que después podrá abrir con un navegador. En este archivo se va a ir llevando la salida de órdenes como `hostname`, `uname`, `uptime`, `free`, `df`, `netstat` y `w` (use la ayuda con **`man`** o **`help`** para ver qué información proporcionan). En la línea 5 se le da valor a la variable `_STAT`

(simplemente el literal `status.html`) y el resto del guion consiste en órdenes que escriben en el archivo cuyo nombre está almacenado en dicha variable.

```
#! /bin/bash
# statgen -- Luke Th. Bullock - Thu Dec 14 11:04:18 MET 2000
#

_STAT=status.html
echo "<html><title>`hostname` Status</title>" > $_STAT
echo "<body bgcolor=white><font color=slategray>" >> $_STAT
echo "<h2>`hostname` status <font size=-1>(updated every 1/2 hour) </h2></font></font>" >>
$_STAT
echo "<pre>" >> $_STAT
echo "<b>Date:</b> `date`" >> $_STAT
echo >> $_STAT
echo "<b>Kernel:</b>" >> $_STAT
uname -a >> $_STAT
echo >> $_STAT
echo "<b>Uptime:</b>" >> $_STAT
uptime >> $_STAT
echo >> $_STAT
echo "<b>Memory Usage:</b>" >> $_STAT
free >> $_STAT
echo >> $_STAT
echo "<b>Disk Usage:</b>" >> $_STAT
df -h >> $_STAT
echo >> $_STAT
echo "<b>TCP connections:</b>" >> $_STAT
netstat -t >> $_STAT
echo >> $_STAT
echo "<b>Users logged in</b> (not showing processes):" >> $_STAT
w -hus >> $_STAT
echo "</pre>" >> $_STAT
```

Para ver el resultado podemos abrir el archivo `status.html` con el navegador. Este tipo de archivo se denomina *CGI* (Common Gateway Interface) que es una tecnología web que permite a un cliente web (navegador) solicitar información de un programa ejecutado en el servidor web.

5.7.3 Adaptar el guion al sistema operativo donde se ejecuta

La orden `uname` nos da el nombre del sistema en uso. Con ella, podemos construir un guion de la forma que se mostrará a continuación con la idea de conseguir el propósito de que nuestro guion emplee unas órdenes u otras dependiendo del sistema operativo en el que nos encontremos.

```
#!/bin/bash

SO=`uname`
case $SO in
       Linux) # ordenes exclusivas de Linux
              echo "Estamos en un SO Linux" ;;
       AIX)   # ordenes exclusivas de AIX
              echo "Estamos en un SO AIX" ;;
       SunOS) # ordenes de Solaris
              echo "Estamos en un SO SunOS" ;;
       *)     # cualquier otro SO
              echo "Sistema Operativo no soportado\n"
              exit 1 ;;
esac
```

5.7.4 Mostrar una raya girando mientras se ejecuta una orden

A continuación vamos a construir un guion bash que muestra una raya girando mientras se ejecuta una orden que consume mucha CPU. En nuestro caso simulamos la orden que consume CPU mediante un bucle `for`, pero esta orden podría ser, por ejemplo, una larga compilación. La idea es que estemos visualizando el giro de una línea

mientras se ejecuta el programa de larga duración para que no pensemos que el sistema no nos está respondiendo.

```
#!/bin/bash
# rotor -- Randal M. Michael – Mastering Unix Shell Scripting, Wiley, 2003
#
function _rotar_linea {
  INTERVAL=1            # Tiempo a dormir entre giro
  TCOUNT="0"            # Para cada TCOUNT la linea gira 45 grados
  while true            # Bucle infinito hasta que terminamos la funcion
  do
    TCOUNT=`expr $TCOUNT + 1` # Incrementa TCOUNT
    case $TCOUNT in
       "1") echo -e "-""\b\c"
            sleep $INTERVAL ;;
       "2") echo -e '\\'"\b\c"
            sleep $INTERVAL ;;
       "3") echo -e "|""\b\c"
            sleep $INTERVAL ;;
       "4") echo -e "/""\b\c"
            sleep $INTERVAL ;;
         *) TCOUNT="0" ;; # Pone a cero TCOUNT
    esac
  done
}
######## Cuerpo principal ############

_rotar_linea &    # Ejecuta la función _rotar_linea en background, es decir,
                  # se ejecuta concurrentemente con el resto del guion.
ROTAR_PID=$!      # Captura el PID del último proceso que está en background.

# Simulamos la ejecución de una orden que consume mucha CPU
# durante la cual mostramos la línea rotando

for ((CONT=1; CONT<400000; CONT++ )) ;
do
   :
done

# Paramos la función _rotar_linea

kill -9 $ROTAR_PID # provoca la terminación del proceso cuyo PID es $ROTAR_PID

# Limpiamos el trazo que queda tras finalizar

echo -e "\b\b"
```

La función `_rotar_linea` se encarga de representar en pantalla la barra que gira; esta función se construye a partir de la línea 4, realizando un bucle infinito que escribe en la pantalla la sucesión de caracteres

```
-\|/-\|/-\|/-\|/...
```

A modo de dibujos animados, al situarlos en el mismo lugar de la pantalla logramos la ilusión óptica del giro de una barra. Observe que en la línea 25 se lanza la ejecución de `_rotar_linea` como actividad "en background" gracias al metacarácter **&**; esto es esencial pues provoca que tengamos dos actividades concurrentes: `_rotar_linea` y la

tarea que corresponde a la ejecución de las instrucciones siguientes que forman parte también del guion. Mientras hemos dejado la barrita girando, en las líneas siguientes se entra en un bucle de larga duración que al terminar deberá parar el movimiento de la barra; para ello se utiliza la orden

```
kill -9 <pid>
```

que finaliza el proceso cuyo pid se pasa como argumento.

Ejercicio 5.6. Copie este ejercicio y pruébelo en su sistema para ver su funcionamiento. ¿Qué podemos modificar para que el giro se vea más rápido o más lento? ¿Qué hace la opción `-e` de las órdenes `echo` del guion?

Consulte la ayuda de bash, observe que aquí hay información muy interesante sobre las órdenes empotradas (*shell builting commands*) como por ejemplo `echo`, y sobre sus secuencias de escape.

Podemos encontrar ejemplos útiles de guiones en diferentes páginas web, entre las que podemos citar la página de Heiner Steven cuya dirección es `http://www.shelldorado.com/scripts/`

5.8 Archivos de configuración

Existen diferentes archivos de configuración que son leídos por el shell cuando se lanza, son los *archivos de arranque*. Estos guiones tienen como objetivo establecer la configuración del shell definiendo variables, funciones, alias, etc.

Los archivos de arranque ejecutados van a depender del tipo de shell. El shell que aparece cuando arrancamos la máquina se denomina *login shell* (desde el que ejecutamos `startx`). Este shell utiliza los archivos:

- `/etc/profile`
- `$HOME/.bash_profile`
- `$HOME/.bash_login`
- `$HOME/.profile`
- `$HOME/.bashrc`
- `$HOME/.bash_logout`

El primero de ellos es un archivo del sistema y, por el directorio donde se encuentra y los permisos asignados, sólo puede ser modificado por el administrador. El resto de archivos de configuración pueden ser modificados por el usuario para adaptar la shell a sus necesidades.

Ejercicio 5.7. Escriba un guion que admita como argumento el nombre de un tipo de shell (por ejemplo, csh, sh, bash, tcsh, etc.) y nos dé un listado ordenado alfabéticamente de los usuarios que tienen dicho tipo de shell por defecto cuando abren un terminal. Dicha información del tipo de shell asignado a un usuario se puede encontrar en el archivo `/etc/passwd`, cuyo contenido está delimitado por `':'`. Cada información situada entre esos delimitadores representa un campo y precisamente el campo que nos interesa se encuentra situado en primer lugar. En definitiva, para quedarnos con lo que aparece justo antes del primer delimitador será útil la orden siguiente:

```
cut -d':' -f1 /etc/passwd
```

Donde la opción `-d` indica cuál es el delimitador utilizado y la opción `-f1` representa a la secuencia de caracteres del primer campo. Realice, utilizando el mecanismo de cauces, el ejercicio pero usando la orden `cat` para mostrar el contenido de un archivo y encauzado con la orden `cut` para filtrar la información que aparece justo antes del delimitador `':'`[3].

Realice también la comprobación de la validez del tipo de Shell que se introduce como argumento. Use para ello la información que encontrará en el archivo `/etc/shells` donde encontrará los tipos de Shell que se pueden utilizar en el sistema.

[3] Utilice la orden `man` para conocer otras posibilidades en el uso de la orden `cut`, en particular, cortar un intervalo de caracteres.

Ejercicio 5.8. Dos órdenes frecuentes de Unix son `tar` y `gzip`. La orden `tar` permite almacenar/extraer varios archivos de otro archivo. Por ejemplo, podemos almacenar el contenido de un directorio en un archivo con

```
tar -cvf archivo.tar directorio
```

(la opción `-x` extrae los archivos de un archivo `.tar`).

La orden `gzip` permite comprimir el contenido de un archivo para que ocupe menos espacio. Por ejemplo, `gzip archivo` comprime `archivo` y lo sustituye por otro con el mismo nombre y con la extensión `.gz`. La orden para descomprimir un archivo `.gz` o `.zip` es `gunzip`.

Dadas estas órdenes construya un guion, denominado `cpback`, que dado un directorio o lista de archivos como argumento(s) los archive y comprima en un archivo con nombre `copiaYYMMDD`, donde `YY` corresponde al año, la `MM` al mes y la `DD` al día, dentro de un directorio denominado `CopiasSeguridad`. El guion debe realizar las comprobaciones oportunas: los argumentos existen, el directorio de destino existe y si no, lo crea.

Ejercicio 5.9. Hacer un script en Bash denominado `newdirfiles` con los siguientes tres argumentos:

- `<dirname>` Nombre del directorio que, en caso de no existir, se debe crear para alojar en él los archivos que se han de crear.
- `<num_files>` Número de archivos que se han de crear.
- `<basefilename>` Será una cadena de caracteres que represente el nombre base de los archivos.

Ese guion debe realizar lo siguiente:

- Comprobar que el número de argumentos es el correcto y que el segundo argumento tenga un valor comprendido entre 1 y 99.
- Crear, en caso de no existir, el directorio dado en el primer argumento a partir del directorio donde se esté situado y que posea permisos de lectura y escritura para el usuario `$USER`.
- Dentro del directorio dado en el primer argumento, crear archivos cuyos contenidos estarán vacíos y cuyos nombres lo formarán el nombre dado como tercer argumento y un número que irá desde 01 hasta el número dado en el segundo argumento.

Práctica 6: Depuración y control de trabajos

6.1 Objetivos principales

- Conocer las principales características de depuración que ofrece bash y saber utilizarlas para detectar errores en un guion bash.
- Conocer los medios que ofrece la shell bash para controlar los trabajos de una sesión.
- Saber utilizar la orden **ps** para conocer el estado de los procesos en ejecución.

Además, se verán las siguientes órdenes:

Órdenes Shell Bash				
jobs	**fg**	**bg**	**%**	**disown**
kill/killall	**wait**	**trap**	**set**	**sleep**
source	**top**	**ps**		

Tabla 6.1. Órdenes de la práctica.

6.2 Características de depuración en bash

La shell bash no contiene ninguna orden específica para depuración de guiones. De hecho los mensajes que se devuelven al detectar un error sintáctico o léxico no suelen ayudar en exceso.

Como muestra, se ha introducido un ligero error en el siguiente script, denominado **pathmas**, que se vio en la sección 3.2 de la práctica 5:

```
#!/bin/bash
# Uso: pathmas directorio [after]
if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
          if ["$2" = "after" ] ; then                    # <<<<<<<<<<<<<<<<<<<<<<<
              PATH=$PATH:$1
           else
              PATH=$1:$PATH
          fi
else
    echo "$1 ya está en el path"
fi
```

Si lo ejecutamos, se obtiene el mensaje:

```
$ ./pathmas: línea 4: [: =: se esperaba un operador unario
```

Ejercicio 6.1. Indique cuál ha sido el error introducido en el guion anterior y cómo se corregiría.

Un guion puede ser erróneo por errores sintácticos, porque no funciona como se desea que lo haga, o porque funcionando correctamente posee efectos secundarios indeseados (bomba lógica). Para depurar un guion defectuoso, se pueden usar las herramientas que se plantean a continuación:

- Usar la orden **`echo`** en puntos críticos del guion para seguir el rastro de las variables más importantes.
- Usar las opciones **`-n`**, **`-v`** y **`-x`** de bash.
- Usar la orden **`trap`** para depuración.
- Usar funciones de "aserción".

6.2.1 Opciones de depuración en bash

El intérprete bash ofrece una serie de opciones que pueden ayudar a detectar el origen de un error. Estas opciones se muestran en la siguiente tabla, donde también se indican las órdenes bash que, insertadas en un guion, activan ese tipo de funcionamiento:

Opción	Efecto	Órdenes equivalentes
`-n`	Chequea errores sintácticos sin ejecutar el guion.	**`set -n`** **`set -o noexec`**
`-v`	Visualiza cada orden antes de ejecutarla.	**`set -v`** **`set -o verbose`**
`-x`	Actúa igual que **`-v`** pero de forma más abreviada. Al visualizar las órdenes, éstas aparecen con todas las sustituciones y expansiones ya realizadas.	**`set -x`** **`set -o xtrace`**

Tabla 6.2. Opciones de depuración en bash.

Estas opciones se pueden usar al invocar bash (pasándoselas como opciones) o bien declarándolas dentro del guion de forma que a partir de la línea donde la activemos se realizará la depuración hasta el final del guion o hasta que se desactive. El chequeo sintáctico, por ejemplo, se activa poniendo en una línea del guion **`set –n`**, y se desactiva (ya no se realizará el chequeo) cuando se encuentre otra línea con **`set +n`**.

Como ejemplo de la primera forma vista, consideremos del siguiente guion al que denominamos **`lee_archivo`**:

```
contador=1
while read lineaLectura ;
do
      echo "Línea $contador: $lineaLectura"
      contador=$[ $contador + 1]
done < archivotest
echo "Finalizado el procesamiento del archivo"
```

Donde el contenido de `archivotest` es el siguiente:

```
Esta es la línea 1 de archivotest
Esta es la segunda línea de archivotest
```

Podemos ver el progreso línea a línea del guion si usamos la opción **`xtrace`** de bash (opción `-x`), como se muestra en el siguiente ejemplo:

```
$ bash -x lee_archivo
+ contador=1
```

```
+ read lineaLectura
+ echo 'Línea 1: Esta es la línea 1 de archivotest'
Línea 1: Esta es la línea 1 de archivotest
+ contador=2
+ read lineaLectura
+ echo 'Línea 2: Esta es la segunda línea de archivotest'
Línea 2: Esta es la segunda línea de archivotest
+ contador=3
+ read lineaLectura
+ echo 'Finalizado el procesamiento del archivo'
Finalizado el procesamiento del archivo
```

El archivo a depurar, en nuestro caso `lee_archivo`, si llevase como primera línea la orden `#!/bin/bash`, se podría observar que dicha línea se ignora durante el proceso de depuración del guion.

En el ejemplo anterior podemos ver cómo `xtrace` inicia cada línea con un `+` (cada nivel de expansión viene representado con un `+`). Este símbolo es personalizable ajustando el valor de la variable empotrada `PS4`. Por ejemplo, podemos definir la variable como `PS4='*** línea $LINENO: '`, donde `$LINENO` representa el número de línea que está leyendo de un archivo, y exportarla previamente a la ejecución del ejemplo anterior.

```
$ export PS4='*** línea $LINENO: '
$ bash -x lee_archivo
*** línea 1: contador=1
*** línea 2: read linea
*** línea 4: echo 'Línea 1: Esta es la línea 1 de archivotest'
Línea 1: Esta es la línea 1 de archivotest
*** línea 5: contador=2
*** línea 2: read linea
*** línea 4: echo 'Línea 2: Esta es la segunda línea de archivotest'
Línea 2: Esta es la segunda línea de archivotest
*** línea 5: contador=3
*** línea 2: read linea
*** línea 7: echo 'Finalizado el procesamiento del archivo'
Finalizado el procesamiento del archivo
```

6.2.2 Realizar la traza con la orden `trap`

La orden **`trap`** sirve para especificar una acción a realizar cuando se recibe una señal. Las señales son en Linux un mecanismo de comunicación entre procesos que permite la notificación de que ha ocurrido un determinado evento (suceso) a los procesos. Para ver su descripción completa utilice `help trap`.

Uno de los usos de esta orden es asistir a la depuración de guiones. Para ello, se invoca esta orden al comienzo del guion pasándole el argumento **`DEBUG`**. Esto hace que la acción especificada como argumento de la orden trap se ejecute después de cada orden de un guion, cuando la opción **`xtrace`** de bash (bash -x) esté activada. Así por ejemplo, si extendemos el guion **`lee_archivo`**, incluyendo un `trap DEBUG` al principio, como se muestra aquí:

```
trap 'echo TRAZA --- contador= $contador lineaLectura= $lineaLectura' DEBUG
contador=1
while read lineaLectura ;
do
      echo "Línea $contador: $lineaLectura"
      contador=$[ $contador + 1]
done < archivotest
echo "Finalizado el procesamiento del archivo"
```

Al invocar el guion con **xtrace** se obtiene:

```
$ export PS4='+ '
$ bash -x lee_archivo
+ trap 'echo TRAZA --- contador= $contador lineaLectura= $lineaLectura' DEBUG
++ echo TRAZA --- contador= lineaLectura=
TRAZA --- contador= lineaLectura=
+ contador=1
++ echo TRAZA --- contador= 1 lineaLectura=
TRAZA --- contador= 1 lineaLectura=
+ read lineaLectura
++ echo TRAZA --- contador= 1 lineaLectura= Esta es la linea 1 de archivotest
TRAZA --- contador= 1 lineaLectura= Esta es la linea 1 de archivotest
+ echo 'Línea 1: Esta es la linea 1 de archivotest'
Línea 1: Esta es la linea 1 de archivotest
++ echo TRAZA --- contador= 1 lineaLectura= Esta es la linea 1 de archivotest
TRAZA --- contador= 1 lineaLectura= Esta es la linea 1 de archivotest
+ contador=2
++ echo TRAZA --- contador= 2 lineaLectura= Esta es la linea 1 de archivotest
TRAZA --- contador= 2 lineaLectura= Esta es la linea 1 de archivotest
+ read lineaLectura
++ echo TRAZA --- contador= 2 lineaLectura= Esta es la segunda linea de archivotest
TRAZA --- contador= 2 lineaLectura= Esta es la segunda linea de archivotest
+ echo 'Línea 2: Esta es la segunda linea de archivotest'
Línea 2: Esta es la segunda linea de archivotest
++ echo TRAZA --- contador= 2 lineaLectura= Esta es la segunda linea de archivotest
TRAZA --- contador= 2 lineaLectura= Esta es la segunda linea de archivotest
+ contador=3
++ echo TRAZA --- contador= 3 lineaLectura= Esta es la segunda linea de archivotest
TRAZA --- contador= 3 lineaLectura= Esta es la segunda linea de archivotest
+ read lineaLectura
++ echo TRAZA --- contador= 3 lineaLectura=
TRAZA --- contador= 3 lineaLectura=
+ echo 'Finalizado el procesamiento del archivo'
Finalizado el procesamiento del archivo
```

Un aspecto importante que se debe recordar sobre `DEBUG` es que no es heredado por las funciones invocadas desde el shell en el que está definido. Por tanto, debemos invocar la declaración "`trap ... DEBUG`" dentro de las funciones en las que deseemos usarlo.

`DEBUG` es un ejemplo de lo que se denomina *señales falsas* (fake signal) que se utilizan en las declaraciones `trap` para que el shell actúe bajo ciertas condiciones. Éstas actúan como las señales, pero son generadas por el propio shell. Otros ejemplos de estas señales son: `EXIT`, `ERR` y `RETURN`. La señal falsa `EXIT` ejecutará su código cuando el shell en la que está activa finalice. Solamente podemos atrapar finalizaciones de un guion (las funciones no generan señales `EXIT`). Por ejemplo, si deseamos que nuestro guion muestre un mensaje cuando finalice (normal o forzadamente) únicamente debemos incluir la línea:

```
trap 'echo Fin del guion $0' EXIT
contador=1
while read lineaLectura ;
do
      echo "Línea $contador: $lineaLectura"
      contador=$[ $contador + 1]
done < archivotest
echo "Finalizado el procesamiento del archivo"
```

La ejecución de nuestro guion de ejemplo `lee_archivo` será:

```
$ bash lee_archivo
Línea 1: Esta es la linea 1 de archivotest
Línea 2: Esta es la segunda linea de archivotest
Finalizado el procesamiento del archivo
Fin del guion lee_archivo
```

La señal `ERR` se activa cuando una orden devuelve un código de finalización distinto de cero (recordemos que las órdenes suelen devolver un valor `0` si ha tenido éxito, es decir, si han funcionado correctamente). Podemos construir una función para imprimir un mensaje cuando alguna orden no finaliza correctamente. Esa función podría ser la siguiente:

```
function _atrapaerror {
    codigo_error=$?       #salvamos el código de error de la última orden
    echo "ERROR línea $1: la orden finalizó con estado $codigo_error"
}
```

La función anterior se usará para que, ante la ocurrencia de un error en la ejecución de cualquier orden, nos indique el código de error correspondiente. Para ello, escribiremos dicha función tal cual en nuestra Shell y, acto seguido, la orden `trap` correspondiente como se muestra a continuación:

```
$ function _atrapaerror {
>     codigo_error=$?       #salvamos el código de error de la última orden
>     echo "ERROR línea $1: la orden finalizó con estado $codigo_error"
> }
$ trap '_atrapaerror $LINENO' ERR
```

Si ejecutamos una orden cuyo resultado dé un error, nos informará la propia Shell de tal error y se ejecutará la función `_atrapaerror` indicando la línea donde se ha producido (el número de línea dependerá de las líneas que se hayan ido introduciendo en la Shell previamente).

```
$ ls archivoquenoexiste
ls: no se puede acceder a archivoquenoexiste: No existe el fichero o el directorio
ERROR línea 11: la orden finalizó con estado 2
```

Por último, `RETURN` se utiliza cuando se regresa tras la ejecución de órdenes o guiones que se han ejecutado con **`source`**. Esta orden permite ejecutar un guion dentro de la Shell actual y no como un proceso aparte, de forma que si se crean o modifican variables permanecerán en la Shell después de la ejecución del guion (ver `help source`).

Para finalizar, indicar que a partir de la versión 3 de la Shell de Bash, existen algunas variables que facilitan la depuración, además de las disponibles `$LINENO` y `$FUNCNAME`, como son: `$BASH_ARGC`, `$BASH_ARGV`, `$BASH_SOURCE`, `$BASH_LINENO`, `$BASH_SUBSHELL`, `$BASH_EXECUTION_STRING`, y `$BASH_COMMAND`. Entre las facilidades que suministra Linux para la depuración, podemos incluir la orden **`script`** que crea una copia de la sesión de un terminal en un archivo para su posterior revisión (ver `info bash`).

Si escribimos la orden `trap` en la Shell nos mostrará el mensaje que aparecería en caso de que aconteciera algún error asociado a alguna señal.

```
$ trap
trap -- '' SIGTSTP
trap -- '' SIGTTIN
trap -- '' SIGTTOU
trap -- '_atrapaerror $LINENO' ERR
```

Si quisiéramos que dejase de informar sobre los errores producidos en la ejecución de órdenes, deberíamos volver a ejecutar la orden trap pero sin mensaje de tratamiento para la señal:

```
$ trap -- '' ERR
```

```
$ trap
trap -- '' SIGTSTP
trap -- '' SIGTTIN
trap -- '' SIGTTOU
trap -- '' ERR
```

Como se puede apreciar, ya no habrá tratamiento de error cuando éste acontezca.

6.2.3 Aserciones

Una función de aserción comprueba una variable o condición en puntos críticos del guion. Por ejemplo, podemos definir una función que comprueba la corrección de una comparación:

```
#!/bin/bash
# asercion.sh
#
_asercion ()                 # Si la condición es falsa, finaliza el guion
{                            # con un mensaje de error
   E_PARAMETRO_ERROR=98
   E_ASERCION_FALLIDA=99
   if [ -z "$2" ] ; then     # Parámetros insuficientes pasados a _asercion
      return $E_PARAMETRO_ERROR
   fi
   lineno=$2
   if [ ! $1 ] ; then
      echo "Falla la aserción: \"$1\""
      echo "Archivo \"$0\", línea $lineno" # da el nombre del guion y núm. de línea
      exit  $E_ASERCION_FALLIDA
      # else retorna y continúa con el guion
   fi
}
a=5
b=4
condicion="$a -lt $b"   # Mensaje de error y salida del guion
                        # Ajustar la variable condicion a otra cosa y ver qué pasa

_asercion "$condicion" $FILENO

# El resto del guion se ejecuta únicamente si la aserción no falla

echo "Si llega aquí es porque la aserción es correcta"
```

En el ejemplo anterior, tal y como se puede comprobar, se define la función `_asercion` que admite dos argumentos. El primero será una cadena de caracteres representativa de una expresión lógica (por esa razón ha de estar entre comillas, pues la expresión `$a –lt $b` tiene espacios en blanco) y el segundo se corresponde a un número de línea representado mediante una cadena de caracteres numérica.

Tras invocar a la función de la aserción, se ejecutan las órdenes en las que asigna valores a una serie de variables locales (`E_PARAMETRO_ERROR` y `E_ASERCION_FALLIDA`). Posteriormente, comprueba si el segundo de los argumentos aportados a la función tiene un nombre formado por, al menos, un carácter (`[ –z "$2" ]`). Al no cumplirse pues el argumento transferido `$FILENO` se corresponde a una variable no inicializada, finaliza la ejecución de la función devolviendo un código de error `$E_PARAMETRO_ERROR`, es decir, ni siquiera realizaría la comprobación pertinente del primero de los argumentos `$condicion`.

Suponiendo que el argumento `$FILENO` estuviese previamente inicializado a algún valor y, por lo tanto, continuase la ejecución de las órdenes de la función `_asercion`, nos encontraríamos con que no se cumple la condición aportada como primer argumento `$a -lt $b`. En ese caso, muestra un mensaje de error correspondiente y abandona no sólo la ejecución de la función sino, además, termina la ejecución del guion.

Ejercicio 6.2. Aplicar las herramientas de depuración vistas en la sección 2 para la detección de errores durante el desarrollo de los guiones propuestos como ejercicios en la práctica 5.

6.3 Control de trabajos en bash

Las órdenes que se mandan ejecutar al shell reciben el nombre de trabajos o jobs. Un trabajo puede estar en primer plano (**foreground**), en segundo plano (**background**) o suspendido (detenido). Durante una sesión, la shell almacena una lista de los trabajos no finalizados. Estos trabajos no finalizados pueden ser aquellos trabajos que se ejecutan en segundo plano o/y los trabajos que han sido suspendidos. Una orden (o trabajo) se ejecuta en segundo plano cuando incluimos un **&** (ampersand) al final de la orden.

El siguiente ejemplo muestra los mensajes que devuelve la shell al ejecutar una orden en segundo plano. La orden que se lanza en segundo plano incluye una llamada a la orden **sleep** para producir una pausa de cinco segundos.

```
$ (sleep 5; echo "Fin de la siesta de 5 segs.")  &
[1] 10217
$ Fin de la siesta

[1]+  Hecho                   ( sleep 5; echo "Fin de la siesta" )
```

El número que aparece entre corchetes ([1]) indica el número de trabajo que acaba de ser lanzado en segundo plano y el siguiente número (10217) representa el identificador de proceso asociado a la orden. Observe que el indicador del sistema (o prompt) se devuelve inmediatamente de forma que se puede introducir otra orden sin que haya finalizado la orden lanzada en segundo plano.

Los trabajos se pueden manipular usando órdenes de la shell. Estas órdenes permiten hacer referencia a un trabajo de varias formas. Una forma de hacerlo es mediante el carácter '`%`', como se muestra en la siguiente tabla:

Especificador	Trabajo que es denotado con dicho especificador
%	Trabajo actual (%+ y %% son sinónimos de este especificador)
%-	Trabajo previo al actual
%n	Trabajo número n
%<cadena>	Trabajo cuya línea de órdenes comienza por <cadena>
%?<cadena>	Trabajo cuya línea de órdenes contiene <cadena>

Tabla 6.3. Especificadores de trabajos.

La tabla siguiente recoge las órdenes más frecuentes de control de trabajos:

Órdenes	Descripción
jobs	Lista los trabajos activos bajo el control del usuario (`help jobs`)
fg	Trae a primer plano un trabajo que se encuentra suspendido o en segundo plano (`help fg`)
bg	Envía a segundo plano un trabajo (`help bg`)
%	Permite cambiar el estado de un trabajo (`help %`)
wait	Espera la finalización de procesos en segundo plano (`help wait`)
disown	Suprime un trabajo de la lista de trabajos activos (`help disown`)
kill	Envía una señal a un/os proceso/s. Por defecto, finaliza la ejecución de un proceso (`man kill`)
ps	Muestra el estado de los procesos actuales en el sistema (`man ps`)
top	Muestra los procesos en ejecución con actualización de su información en tiempo real (`man top`)

Tabla 6.4. Órdenes de control de trabajos.

6.3.1 La orden `jobs`

La orden **jobs** permite ver el estado de los trabajos suspendidos y en segundo plano. La opción -l permite visualizar, junto con la información normal, el identificador de proceso asociado al trabajo. A continuación, se muestran algunos ejemplos de utilización de la orden **jobs**:

El siguiente ejemplo muestra cómo ver la lista de los trabajos activos:

```
$ sleep 20 &
$ nano prueba
<CTRL>-Z              #Suspendemos la ejecución de nano
$ xterm &
$ jobs -l
[1]  22852 Ejecutando              sleep 20 &
[2]+ 22853 Detenido (señal)       nano prueba
[3]- 22854 Ejecutando              xterm &
```

Como se comprueba en el ejemplo, introducir el carácter de detención (generalmente <CTRL>-Z) mientras un trabajo interactivo se ejecuta, provoca que el proceso sea parado y se devuelva el control al shell.

6.3.2 Las órdenes `fg, bg y %`

La orden **fg** seguida de un especificador trae a primer plano el trabajo especificado (trabajo de fondo o suspendido). Esta orden, usada sin argumentos, lleva el trabajo actual a primer plano.

El siguiente ejemplo ilustra el uso de dicha orden:

```
$ jobs -l
[1]-  3289 Ejecutando              sleep 20 &
[2]+  3290 Ejecutando              xterm &
$ fg %1                 #Ponemos el trabajo 1 (sleep 20) en primer plano (esperamos)
sleep 20
$ jobs
[2]+  Ejecutando              xterm &
$ jobs -l
[2]+  3290 Ejecutando              xterm &
$ fg %2                    #Ponemos el trabajo 2 (xterm) en primer plano
xterm                      #Finalizamos la sesión con xterm y volvemos
```

La orden **bg** es la opuesta de **fg**. Esta orden seguida de un especificador envía a segundo plano el trabajo especificado. Esta orden, usada sin argumentos, lleva el trabajo actual a segundo plano (background).

Para ilustrar el uso de esta orden, el siguiente ejemplo muestra cómo se suspende un trabajo interactivo (edición de textos) y se pone en segundo plano para poder lanzar otros trabajos y después se pone el trabajo interactivo en primer plano para seguir con el mismo.

```
$ nano prueba
<CTRL>-Z              #Suspendemos la ejecución de nano
Use "fg" para volver a nano
[1]+  Detenido                nano prueba
$ jobs
[1]+  Detenido                nano prueba
$ bg %1
[1]+ nano prueba &

# Ponemos nano en segundo plano. Aunque no lo parezca, lo está. Pulse enter y podrá
# comprobar que aparece el prompt del terminal.

# Realizamos varias tareas en esta Shell, por ejemplo ls -l, cat /etc/passwd, etc.

$ fg      #Volvemos a trabajar con nano (en primer plano)
```

La orden `%` lleva a primer plano al trabajo actual, si no tiene argumento, o al trabajo especificado por el argumento. Si está presente el `&`, envía el trabajo a segundo plano. Esta orden es una abreviación de `fg` y `bg`. En los ejemplos anteriores podríamos haber sustituido `fg` y `bg` por el `%`. Probad esta parte.

6.3.3 Esperando a procesos en segundo plano

La orden **wait** produce una espera hasta que el trabajo especificado como argumento haya finalizado su ejecución. El argumento también puede ser un identificador de proceso (`PID`), en cuyo caso, se esperaría la finalización de dicho proceso. Sin argumentos, la orden produce una espera hasta que todos los trabajos en segundo plano hayan terminado su ejecución. El siguiente ejemplo muestra su funcionamiento:

```
$ (sleep 50; echo "Fin de la siesta de 50 segs.") &
[1] 3382
$ (sleep 40; echo "Fin de la siesta de 40 segs.") &
[2] 3384
$ (sleep 20; echo "Fin de la siesta de 20 segs.") &
```

```
[3] 3388
$ wait %3           # Espero a que acabe el trabajo 3 (la siesta de 20 segs.)
Fin de la siesta de 20 segs.
[3]+  Hecho                   ( sleep 20; echo "Fin de la siesta de 20 segs." )
$ jobs -l
[1]-  3382 Ejecutando            ( sleep 50; echo "Fin de la siesta de 50 segs." ) &
[2]+  3384 Ejecutando            ( sleep 40; echo "Fin de la siesta de 40 segs." ) &
$ wait              # Espero a que acaben los trabajos restantes
Fin de la siesta de 40 segs.
Fin de la siesta de 50 segs.
[1]-  Hecho                   ( sleep 50; echo "Fin de la siesta de 50 segs." )
[2]+  Hecho                   ( sleep 40; echo "Fin de la siesta de 40 segs." )
```

6.3.4 Eliminando procesos con las órdenes `disown` y `kill/killall`

La orden **`disown`** elimina el trabajo cuyo identificador se da como argumento siempre y cuando dicho trabajo esté activo, es decir, no tiene efecto sobre trabajos que estén suspendidos (detenidos). Si no se aportasen argumentos, suprimiría el trabajo actual. Examina las opciones de **`disown`** usando la orden **`man`**.

El siguiente ejemplo muestra cómo eliminar selectivamente trabajos en una sesión usando diferentes opciones de la orden **`disown`**:

```
$ jobs
[1]+  Detenido                nano prueba.txt
[2]   Ejecutando              sleep 300 &
[3]-  Ejecutando              xterm &
$ disown %2      # Elimino el trabajo 2
$ jobs
[1]+  Detenido                nano prueba.txt
[3]-  Ejecutando              xterm &
$ disown -r      # Elimino los trabajos en ejecución
$ jobs
[1]   Detenido                nano prueba.txt
$ disown -a      # Elimino todos los trabajos restantes en la lista
bash: aviso: borrando el trabajo detenido 1 con grupo de proceso 3451
```

La orden **`kill`** también permite eliminar procesos, pero es mucho más general que **`disown`** ya que actúa tanto sobre procesos activos como suspendidos. La orden **`kill`** sirve para enviar a un proceso una señal. La acción por omisión al ejecutar la orden **`kill`** es finalizar el proceso (enviar la señal **`SIGTERM`**) o procesos indicados con identificadores de procesos o especificadores de trabajo (ver opción **`-s`** de la orden **`kill`**).

Para ilustrar el uso de **`kill`** en la terminación de procesos se muestra el siguiente ejemplo:

```
$ jobs -l
[3]  16478 Ejecutando              xterm &
[4]+ 16506 Detenido (señal)       nano prueba.txt
[5]  16507 Ejecutando              sleep 300 &
[6]  16508 Ejecutando              xterm &
[7]- 16525 Ejecutando              firefox &
$ kill %5   # Finalizo el trabajo 5
$ jobs
[3]   Ejecutando              xterm &
```

```
[4]+  Detenido                nano prueba.txt
[5]   Terminado               sleep 300
[6]   Ejecutando              xterm &
[7]-  Ejecutando              firefox &
$ kill 16478 16508   # Finalizo los procesos 16478 y 16508
$ jobs
[3]   Salida 15               xterm
[4]+  Detenido                nano prueba.txt
[6]   Salida 15               xterm
[7]-  Ejecutando              firefox &
$
```

A veces la señal que envía **kill** a un proceso puede no finalizarlo debido a que el proceso la ignore o realice una acción distinta a la especificada por defecto. Para forzar la terminación de un proceso en estas circunstancias se debe invocar la orden **kill** con la opción `-9`. Revise las opciones más importantes de las órdenes **kill** y **killall** mediante el uso de **man** y compruebe las diferencias entre ambas órdenes.

6.3.5 Examinando el estado de los procesos con `ps`

Para obtener una información más detallada sobre los procesos del sistema se puede usar la orden **ps**. Esta orden muestra un listado de información sobre los procesos actuales y tiene muchas opciones.

Usada la orden sin argumentos, **ps** muestra el listado de información sobre los procesos de la sesión actual. También es posible obtener la información sobre el estado de procesos concretos dando sus identificadores como argumento. El siguiente ejemplo muestra los usos anteriores:

```
$ ps                 # Muestra el estado de los procesos de la sesión
  PID TTY          TIME CMD
 3743 pts/0    00:00:00 bash
 3906 pts/0    00:00:00 nano
 3909 pts/0    00:00:00 xterm
 3911 pts/0    00:00:00 firefox
 3916 pts/0    00:00:00 run-mozilla.sh
 3920 pts/0    00:00:01 firefox-bin
 3935 pts/0    00:00:00 ps
$ ps 3911 3920      # Muestra el estado del proceso 3911 y 3920
 PID  TTY      STAT   TIME COMMAND
 3911 pts/0    S      0:00 /bin/sh /usr/lib/firefox-3.6.12/firefox
 3920 pts/0    Sl     0:01 /usr/lib/firefox-3.6.12/firefox-bin
```

Las opciones más comunes de esta orden son:

Opción	Efecto
-e	Muestra información de todos los procesos
-l	Muestra la información sobre los procesos en formato largo
-u <nombre>	Muestra el estado de los procesos del usuario <nombre>
-o <formato>	Permite usar un formato definido por el usuario para el listado

Tabla 6.5. Opciones de la orden **ps**.

Se recomienda usar la orden **man** para conocer más sobre sus múltiples opciones y el significado de la información que nos muestra.

El siguiente ejemplo ilustra el uso de una de las opciones más útiles de **ps.** La opción **-lu** visualiza el estado de los procesos en ejecución en formato largo para un usuario concreto indicándose mediante el nombre concreto o variable que identifica el número asignado por el sistema (por ejemplo, es posible usar la variable `$UID` para referirnos a nuestro usuario):

```
$ ps -lu jmantas
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
1 S  1000 11166     1  0  80   0 - 21276 poll_s ?        00:00:00 gnome-keyring-d
4 S  1000 11184  9360  0  80   0 - 41992 poll_s ?        00:00:00 gnome-session
1 S  1000 11218 11184  0  80   0 -  2984 poll_s ?        00:00:00 ssh-agent
1 S  1000 11221     1  0  80   0 -  6565 poll_s ?        00:00:00 dbus-launch
1 S  1000 11222     1  0  80   0 -  6016 poll_s ?        00:00:00 dbus-daemon
…
0 R  1000 17245 16238  0  80   0 -  3814 -      pts/0    00:00:00 ps
```

El siguiente ejemplo muestra cómo visualizar, para los procesos de la sesión actual, su identificador de proceso, el nombre corto del ejecutable y el consumo de memoria en kilobytes.

```
$ ps -o pid,cmd,size
PID  CMD                          SZ
3743 bash                        568
3906 nano                        652
4042 ps -o pid,cmd,size          612
```

6.3.6 Examinando los procesos en ejecución con `top`

La orden **top** muestra una lista de procesos en ejecución de forma actualizada en tiempo real. La lista mostrará los procesos ordenados en función del consumo de CPU e información descriptiva de los mismos como el `PID`, usuario al que pertenece, consumo de CPU, consumo de memoria RAM, etc. Resulta una orden muy útil para administrar sistemas UNIX/Linux pues es capaz de mostrar aquellos usuarios que consumen cierta cantidad de recursos esenciales en un momento dado.

```
$ top
top - 22:07:40 up 2 min,  1 user,  load average: 6.75, 3.23, 1.22
Tasks: 133 total,   1 running, 132 sleeping,   0 stopped,   0 zombie
Cpu(s): 15.5%us, 73.6%sy,  0.0%ni,  3.2%id,  5.7%wa,  0.0%hi,  2.1%si,  0.0%st
Mem:   1026080k total,   427788k used,   598292k free,    47316k buffers
Swap:  1046524k total,        0k used,  1046524k free,   177084k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1907 jrevelle  20   0  2816 1132  848 R  3.7  0.1   0:00.05 top
   18 root      20   0     0    0    0 S  1.8  0.0   0:01.42 kworker/0:1
  881 root      20   0 97828  54m 4960 S  1.8  5.5   0:13.82 Xorg
 1779 jrevelle  20   0 11068 3076 2636 S  1.8  0.3   0:00.07 gvfsd-trash
    1 root      20   0  3316 1872 1280 S  0.0  0.2   0:03.80 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.05 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.10 ksoftirqd/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kworker/0:0
    5 root      20   0     0    0    0 S  0.0  0.0   0:01.78 kworker/u:0
    6 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
. . . . . .
```

Use **man top** para conocer las distintas formas de uso de esta orden de gestión de procesos activos.

Ejercicio 6.3. Escribir un guion que nos dé el nombre del proceso del sistema que consume más memoria.

Ejercicio 6.4. Escribir un guion que escriba números desde el 1 en adelante en intervalos de un segundo ¿Cómo se podría, desde otro terminal, detener la ejecución de dicho proceso, reanudarlo y terminar definitivamente su ejecución?

Ejercicio 6.5. ¿Se puede matar un proceso que se encuentra suspendido? En su caso, ¿cómo?

Ejercicio 6.6. ¿Qué debemos hacer a la orden `top` para que nos muestre sólo los procesos nuestros?