ALGUNOS CONCEPTOS BÁSICOS

- Visión panorámica
- Variables de entorno
- Directorios
- Comandos
- El Núcleo del sistema (kernel)
- Procesos
- Test

Visión panorámica:
En su momento trataremos los temas abordados en este capítulo de forma completa. En este momento dado que partimos de cero se hace necesario al menos esbozar una serie de conceptos que son tan elementales que dificilmente se puede explicar algo sin hacer referencia a ellos.

Por otra parte estos conceptos están interrelacionados de tal forma que tampoco se puede abordar ninguno de ellos en detalle en este momento. Por ello nuestro primer objetivo es ofrecer una visión panorámica del SO. Por borrosa que resulte siempre será mejor esto que empezar directamente abordando en detalle los temas.

Usaremos muchas veces de forma indistinta Unix y Linux. No son cosas equivalentes pero en este curso consideraremos ambas cosas como equivalentes salvo que concretemos detalles específicos de Linux. Para nosotros Linux es un SO tipo Unix y precisamente nos centraremos en los aspectos más generales de estos sistemas operativos. Entre sus características más destacables está la de ser un SO multitarea y multiusuario. Un sistema multitarea es aquel que puede ejecutar varios procesos simultaneamente. Para ello se puede usar uno o mas procesadores físicos. En el caso de un solo procesador lo que ocurre es que el tiempo del procesador se va repartiendo para atender los distintos procesos creando la ilusión de que todo ocurre simultaneamente. Un sistema multiusuario es aquel que está pensado para ser utilizado por varios usuarios simultaneamente. En la práctica un sistema multiusuario requiere capacidad multitarea. En el caso concreto de Linux es además un SO multiplataforma ya que puede funcionar en diferentes arquitecturas.

Los conceptos que mencionemos en este curso procuraremos ilustrarlos con ejemplos.

El comando 'echo' es un comando que vamos a usar mucho para practicar. Este comando se limita a visualizar en pantalla todo aquello que se le pasa como argumentos. Por eso nos resultará muy util.

Lo primero que vamos a practicar y que no debe olvidar es el hecho de que en Unix los ficheros, comandos etc.. deben indicarse exactamente respetando la diferencia entre mayúsculas y minúsculas.

```
$ echo hola
$ ECHO hola
```

La segunda linea habrá producido un mensaje de error porque no existe nigún comando llamado ECHO con mayúsculas.

Variables de entorno:
Empezaremos hablando de las variables sin profundizar ahora en ello. Una variable solo es un elemento que tiene un nombre y que es capaz de guardar un valor. Para definir una variable basta poner su nombre un igual y su valor. (Ojo no dejar espacios).

```
$ VAR33=valor_de_la_variable_VAR33
```

En Unix las variables del sistema se llaman variables de entorno. Para consultar el valor de una variable se utiliza el nombre de la variable precedido por '$'.

```
$ echo $VAR33
```

Para ver todas las variables y sus valores se utiliza el comando set.

Para hacer la variable exportable se usa el comando 'export'. Se puede hacer que una variable sea de solo lectura con el comando 'readonly'. Este comando sin parámetros mostrará todas las variables que son de solo lectura. Pruebe estos comandos:

```
$ set
$ readonly
$ export
```

Existen algunas variables predefinidas y hay tres variables importantes que mencionaremos en seguida y que son $PATH, $PS1 y $PS2.

Directorios:
Aquí también tenemos que hacer un pequeño avance sobre la estructura de directorios de Unix porque si no puede que no se entienda lo que vamos a hablar sobre la variable $PATH.

En Unix la estructura de directorios es en forma de arbol similar a la de Msdos. Dado que la estructura de directorios se organiza como las ramas de un arbol para localizar un punto cualquiera hay que utilizar cada uno de los directorios que conducen a ese punto desde el directorio raiz. Nosotros lo llamaremos camino y en Unix se le suele llamar path. Se empieza en el directorio raiz representado por '/' y se avanza por las ramas de ese arbol separando cada identificador por un nuevo caracter '/'. De esta forma '/usr/local/bin' indica un lugar concreto en el arbol de directorios. Quizás se pregunte porqué Unix usa '/' en lugar de '\' como en Msdos. Recuerde que Msdos fué posterior a Unix y que a Bill Gates le gusta inventar cosas totalmente nuevas y revolucionarias. Aunque el astuto lector ya se habrá dado cuenta, advierto a los despistados que estoy ironizando.

Aprovechando que estamos hablando de directorios mencionaremos que hay directorios con significado especial. Está el directorio raiz '/' que ya hemos mencionado. Está el directorio 'home' que es el punto donde el sistema nos situa para trabajar cuando entramos en él. Recordemos que en Unix normalemente no disponemos de todo el sistema para nosotrso solos. Luego podemos cambiar de directorio de trabajo. El directorio de trabajo actual se representa como directorio '.', El directorio anterior o directorio padre de este directorio se representa por '..'. Los comandos 'cd', 'mkdir' y 'pwd' sirven para cambiar de directorio actual, crear un directorio y averiguar en que directorio nos encontramos actualmente.

Estamos dando ideas en forma intuitiva. En realidad cuando digo " ... en que directorio nos encontramos actualmente ..." es una forma de expresarse. Normalmente el usuario no estará en un directorio sino sentado en una silla sentado delante de una pantalla, pero la shell mantiene en todo momento algún directorio abierto como directorio de trabajo actual.

Comandos
Un comando es generalmente un fichero ejecutable localizado en alguna parte de nuestro sistema. Existe una variable llamada $PATH que contiene una lista de caminos de busqueda para los comandos todos ellos separados por ':'. Compruebe el valor de su variable $PATH.

```
$ echo $PATH
```

Si un comando no está localizado en ninguno de esos caminos deberá ser referenciado indicando el camino completo ya que de otra forma no podrá ser referenciado. Si quiere saber donde se encuentra un ejecutable use el comando which. Este comando buscará en los caminos contenidos en la variable $PATH hasta que encuentre el comando y entonces mostrará el camino completo que conduce al comando. Puede ocurrir que un comando se encuentre duplicado y que los dos sitios donde se encuentre figuren en el $PATH. En ese caso se ejecutará el comando que se encuentre en el primer camino referenciado en el $PATH. Igualmente el comando which solo mostrará el primer camino referenciado en el $PATH. Estamos usando un comando llamado echo para mostrar valores. Vamos a localizarlo.

```
$ which echo
```

Echo esta situado en uno de los caminos contenidos en $PATH. En resumidas cuentas $PATH es una variable que usa el interprete de comandos para localizar los comando. En la lección siguiente hablaremos de 'man'. Es un comando que sirve para consultar el manual en linea de Linux. Pues bien existe una variable llamada $MANPATH que contiene la lista de los caminos donde el comando 'man' debe buscar las páginas del manual. La variable $MANPATH será usada por el comando 'man' y quizas algún otro. La variable $PATH será usada por la shell y quizas por algún otro comando como por ejemplo 'which' del cual acabamos de hablar.

Somos conscientes de que no estamos explicando gran cosa ahora, pero son conceptos muy elementales que vamos a utilizar antes de explicar en profundidad todo el sistema de ficheros de Unix.

Más adelante también hablaremos más detenidamente sobre las variables de entorno. Estamos dando unos pequeños esbozos sobre algunas cuestiones porque hay mucha interrelación de unos conceptos con otros y resulta imposible abordar nada en profundidad al principio.

Usuarios
Linux es un sistema operativo multiusuario y eso implica una filosofía de uso muy distinta a la del tradicional ordenador personal. Cuando un usuario va a usar un SO Tipo Unix lo primero que necesita hacer es identificarse para ser autorizado a abrir una sesión de trabajo. También es multitaréa y por ello en el mismo instante varios procesos pueden estar funcionando y cada uno puede pertenercer a usuarios distintos. La información que se guarda en el disco duro también puede pertenecer a distintos usuarios y para evitar que todo ello provoque conflictos existen unos atributos de usuario que se asocian a los ficheros a los directorios, a los procesos, etc. En función de esto hay cosas que estarán permitidas a ciertos usuarios y a otros no.

Los usarios pueden estar organizados en diferentes grupos a fin de poder manejar permisos a nivel de grupo. Esto se hace para simplificar la administración del sistema.

Los usuarios y los grupos dentro del sistema se manejan como un número llamado UID y GID respectivamente. Los números en un ordenador se manejan con mucha más facilidad que un literal. Por eso el nombre de usuario y el nombre de grupo se guardarán solo en un fichero junto al número UID y GID asociado,

pero para el sistema un usuario, un grupo, un proceso, un fichero, y muchas otras cosas se identifican por una clave numérica. Para un ordenador resulta más sencillo consultar si el proceso 456 perteneciente al usuario 105 tiene permiso de escritura en el fichero 48964 que consultar si el usuario 'pepito' perteneciente al grupo 'alumnos' tiene permiso de escritura en el fichero '/home/pepito/leccion005.txt'. Cuando el ordenador tiene que mostrar información en formato inteligible buscará la descripción correspondiente a cada clave numérica. Cada clave recibe un nombre como por ejemplo los UID, GID que ya hemos comentado otra sería por ejemplo el PID, para procesos y existen otras muchas más que iremos aprendiendo y que forman parte de la jerga de este SO.

Para ver los datos de identificación relativos a su usuario pruebe lo siguiente:

```
$ id
```

El comando 'id -un' o el comando 'whoami' muestran su nombre de usuario.

```
$ whoami
```

Existe un usuario especial que goza absolutamente de todos los privilegios y que se llama root. Su número de usuario es decir su UID es 0.

El Núcleo del sistema (kernel)

El núcleo del sistema llamado también kernel es el encargado de realizar la mayoría de funciones básicas del sistema y gestiona entre otras cosas la memoria, los ficheros, los usuarios, las comunicaciones, los procesos, etc. La gestíon de estas cosas se hacen por medio de un limitado número de funciones que se denominan llamadas al sistema y que pueden ser usadas por los programas. Los procesos que usan una llamada al sistema cambian su modo de ejecución. Mientras están ejecutando la llamada del núcleo se dice que estan en modo núcleo y cuando están ejecutando código que no pertenece al núcleo se dice que están en modo usuario. Son dos niveles de ejecución distintos ya que el modo núcleo es un modo privilegiado. Esto garantiza a nivel de hardware que ningún programa de usuario pueda acceder a recursos generales del sistema ni interactuar con otros procesos a no ser que use las llamadas del núcleo las cuales establecerán si tiene o no permiso para hacer ciertas cosas. Esto proporciona gran robustez de funcionamiento. Un programa mal diseñado no perjudicará jamás al sistema ni a otros procesos. Cada proceso tiene su propia zona de memoria y no puede acceder fuera de ella ni intencionadamente ni accidentalmente. Para que un programa pudiera tener un efecto destructivo en el sistema tendría que pertenecer a 'root' o pertenecer al propio nucleo del sistema y solo el admiministrador 'root' puede alterar el dicho nucleo. Si el ordenador aparece un buen día destrozado a martillazos también buscaran la forma de culpar a 'root' para no perder la costumbre.

Procesos

Un proceso a diferencia de un programa es algo vivo es decir algo que está funcionando. En un sitema multitarea como este, un programa puede dar lugar a varios procesos. A cada proceso le corresponderá con un número de identificación llamado PID que le identifica totalmente. Además de esto se guarda la información de identificación del usuario propietario. Cuando un usuario ejecuta un comando se arranca el proceso correspondiende del cual generalmente permanecerá como propietario. Es decir el sistema no acepta órdenes anónimas. Siempre figurará un usuario para hacer cada cosa. Esto se indica con un número UID para identificar el usuario correspondiente. No siempre este UID se corresponde con el usuario que arrancaro el proceso. Por ello existe además de un UID un identificador de usuario efectivo (EUID) que es el que realmente es tenido encuenta por el sistema a la hora de conceder permiso para hacer ciertas cosas. El EUID de 'root' es 0. Ahora no importa como pero algunos comandos de forma bien controlada podrían convertirnos virtualmente en

superusuarios haciendo que su EUID valga 0. Esto serviría por ejemplo para permitir hacer cosas especiales y muy concretas nada más que en condiciones normales solo 'root' podría hacer. No hemos mencionado intencionadamente algunas cosas (como por ejemplo el grupo de usuario), porque estamos simplificando mucho intencionadamente para intentar que en este momento solo capte una primera idea general.

Si se siente un poco mareado y confuso relajese y acostumbrese a esta sensación. Los primeros pasos suelen ser los más complicados. Especialmente en Unix porque no fué diseñado como un sistema intuitivo.

El manual del sistema

man(1), apropos(1)

- Gereralidades del manual
- Secciones del manual
- Conclusiones
- Test

Gereralidades del manual
Man es el manual en linea de todos los sistemas operativos tipo Unix. Esta no es la lección más atractiva de este curso pero si que es una de las más necesarias. Uno de los objetivos del curso es alcanzar cierto nivel de autosuficiencia en Linux. Man no es la única fuente de información pero frecuentemente suele ser el primer sitio donde se suele mirar. Si usted hace una consulta en una lista de usuarios de internet de Linux sobre un tema que podría haber averiguado por si mismo consultando el manual alguien le recordará (en tono más o menos amable dependiendo del día que tenga) que esa información estaba en el manual.

Se pueden explicar muchas cosas sobre man pero es imprescindible que practique usted. Para ello de un rápido primer vistazo a la página man relativa al comando man.

Mientras no indiquemos lo contrario conviene que pruebe todos los comandos que se indican a modo de ejemplo. Si tiene posibilidad de usar dos sesiones le resultará más comodo practicar sobre la marcha. Teclee ahora el comando siguiente:

```
$ man man
```

Habrá visto un documento que explica el uso de man y que tiene una estructura característica de todas las páginas man.

Las páginas del manual utilizan un formateador de documentos llamado troff. Permite especificar el estilo de un documento. (Manera en que se ven los títulos, encabezados, parrafos, donde aparecen los números de página, etc. Debido a su flexibilidad troff resulta bastante dificil de usar. nroff y groff sirven para lo mismo pero no explicaremos sus diferencias. Pensamos que solo necesitar conocer su existencia. Si no instala estos programas no podrá usar man en su ordenador.

Otro ejemplo. Para obtener toda la información relativa al comando 'ls' haga lo siguiente:

```
$ man ls
```

Cuando no se conoce exactamente la palabra que se desea buscar se puede buscar por palabra clave. Para ello usaremos la opción -k, y la opción -f. Esta opción no está disponible en muchos sistemas debido a que hay que generar referencias y esto consume mucho espacio en disco. El administrador puede generar las referencias usando el comando 'catman'

Cuando no conocemos el nombre de un comando y tampoco encontramos nada por palabra clave pero sabemos que es lo que hace podemos usar el comando 'apropos'. Compruebelo tecleando los siguientes comandos:

```
$ apropos man
$ apropos apropos
$ man apropos
```

Si tiene su sistema Linux correctamente configurado para imprimir documentos en formato PostScrip podrá imprimir una página del manual haciendo

```
$ man -t man | lpr
```

Secciones del manual

Para indicar en un documento una referencia a una página del manual se suele indicar con el comando seguido de la sección entre parentesis. El título de esta lección 'man(1)' es un ejemplo. Para consultar un comando en una sección concreta habría que teclear man <numero_seccion> <comando_o_funcion>. Veamos un par de ejemplos con printf(1) y printf(3).

```
$ man 1 printf
$ man 3 printf
```

Como habrá podido observar se refieren a cosas distintas. En el primer caso printf es un comando y en el segundo una funcion estandar de C con el mismo nombre. Si no se indica la sección 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. Por el contrario usando la opción -a presentará, secuencialmente, todas las páginas disponibles en el manual. Compruebe esto con los siguientes ejemplos:

```
$ man printf
$ man -a printf
```

Consute las secciones que están disponibles en su sistema en **man(1)**. Acabamos de indicar lo que significa man(1) así que el astuto lector ya debería estar mirando en la sección 1 del manual.

Conclusiones

Para finalizar también mencionaremos otra fuente importante de consulta en Linux que son los HOWTOs y los FAQs. Estos son documentos que explican como hacer algo o las preguntas más frecuentes relativas a un

determinado tema. Este tipo de documentos monotemáticos son frecuentes en Linux. Proceden de contribuciones desinteresadas de usuarios bien informados y están disponibles en distintos formatos (HTML. texto plano, etc). Suelen ser más didácticos que las páginas del man pero man es la referencia obligada para cada comando y cada función importante. Nos evita tener que recordar todas las opciones de cada comando ya que frecuentemente cada comando tiene un amplio juego de opciones. Hay algunos documentos en español especialmente adecuados para novatos. Puede localizar estos documentos en el rincón del novato .

Hemos visto unas pocas opciones del comando man. Intente utilizar algunas otras opciones. Es importante que se familiarice con él. Las páginas man no están pensadas para enseñar, pero cada vez que tenga una duda de como se utiliza algo tendrá que recurrir a ellas como primera fuente de información.

Con esto pretendemos que abandone usted algunas de sus viejas costumbres. Antes usted pensaba de la siguiente manera:

Si funciona aceptablemente para que tocarlo.
Si no funciona apagamos y volvemos a encender
Si nada de esto sirve llamamos a un amigo.

Ahora ya sabe que también puede optar por una lectura del manual. Hagalo pero consulte de momento solo la información relativa a los números de sección, partes de un manual, y las opciones -a, -k, -t. Asumimos que no es necesario la lectura en profundidad porque habrá cosas que aun no puede interpretar. Tenga en cuenta que partimos de un nivel cero. Aún no sabe usted lo que significan algunas cosas. Por eso una vez finalizado el curso convendrá que vuelva a mirar esta página man relativa al comando man para sacarle más prLA SHELL

sh(1) ksh(1) csh(1) bash(1)

- Introducción a la shell de Unix
- Estructura de la línea de orden
- Expansión de la línea de ordenes
- Redirección de entrada salida
- Operador grave
- Caracteres de escape
- Resumen
- Test

Introducción a la shell de Unix
Existen varias shells para Unix, Korn-Shell (ksh), Bourne-Shell (sh), C-Shell (csh), y muchas más. Existen algunas para propósitos especiales. Por ejemplo la remote-Shell (rsh) se utiliza para ejecutar comandos en un ordenador remoto. La Secure Shell (Ssh) se utiliza para establecer una conexion segura con un ordenador remoto.

La más utilizada en Linux es la Bourne-Again SHell (bash).

Nosotros de momento vamos a tratar principalmente la Bourne Shell que es la más estándar.
La Korn-Shell y la Bash son distintos superconjuntos distintos de la Bourne-Shell y por ello todo lo que se diga para la Burne-Shell será válido también para la Korn-Shell y para la Bash.

En Linux se suele usar la Bourne-Again SHell (bash), como sustituta de la Bourne-Shell (sh). Puntualmente también explicaremos alguna peculiaridad de la bash.

Para saber que shell está usando usted haga lo siguiente:

```
$ ps | grep $$
```

Si aparece -bash o -sh puede continuar sin problemas ya que está usando una shell adecuada para practicar lo que viene a continuación. En caso contrario tecle el comando 'sh' o el comando 'bash' antes de continuar. Vuelva a realizar la comprobación anterior y verá que ahora esta usando otra shell.

En cualquier caso cualquier Shell es un programa normal y corriente, pero incorpora muchos de los conceptos más prácticos de Unix. No tiene nada de particular que algunos sistemas incorporen algunas Shells distintas.

Una Shell no es solo un interprete de comandos. Una Shell es sobre todo un interprete de un potente lenguaje.

Estructura de la linea de orden
Sabe usted lo que es un introductor ? (en ingles lo llaman prompt). Pues es aquello que el interprete de comandos muestra para indicar que está esperando a que se introduzca una orden. En Unix el introductor de la orden de comandos no es siempre el mismo. Por defecto suele venir configurado distintos introductores para distintos interpretes de comandos y también se usa un introductor distinto para el usuario root. Sin embargo el introductor puede ser variado ya que es almacenado en una variable del sistema. En realidad la shell utiliza dos introductores distintos. Para ver cuales está utilizando ahora teclee lo siguiente:

```
$ echo "Introductor 1=$PS1"
$ echo "Introductor 2=$PS2"
```

Cuando aparece el primer introductor del sistema $PS1 indica que la shell está esperando la introducción de una orden. Las ordenes se terminan mediante . Si después de pulsar la shell no considera que el comando este completo quedará esperando más entrada mostrando el segundo introductor $PS2.

Si alguna vez no es capaz de terminar la introducción de un comando pruebe a abortar usando <Ctrl-C>

Una orden constará de un número variable de elementos separados por blancos, o por <tab>.

En una orden se pueden distinguir comandos, opciones, argumentos, meta-caracteres, comentarios, comandos internos...etc. Los blancos se usan para separar las opciones y argumentos presentes en una linea de ordenes y usados de esta forma (como separadores) una secuencia de blancos tiene el mismo efecto que uno solo. (Ojo en Unix las mayúsculas y minúsculas son significativas.)

A continuación se menciona de forma no exaustiva los elementos del lenguaje shell-script. No intentamos que se aprenda todos estos elementos ni vamos a comentar ahora todos ellos. Bastará que se haga una idea de que tipos de elementos pueden intervenir.

- **Comandos:**
  Son ficheros ejecutables. Para que la shell localice el comando deberá estar en un subdirectorio que forme parte de la variable PATH o de lo contrario debe especificarse el camino completo.

- **Opciones:**
  Generalmente las opciones de un comando son letras precedidas de un signo '-'. En algunos comandos se pueden poner varias opciones como varias letras seguidas precedidas del signo '-'. Algunas opciones pueden venir como '--<opcion>' y concretamente en Linux es muy frecuente poder usar las opciones --help y --version en casi todos los comandos. Precisamente con la opción --help obtendremos generalmente la lista de opciones disponibles para un comando. También es frecuente el uso de opciones precedidas por '+'. En algunos comandos el orden de las opciones es muy significativo. No merece la pena hablar más de esto ahora porque vamos a usar un montón de comandos con un montón de opciones en este curso y tendremos ocasión de practicar mucho con ellas.
- **Meta-caracteres :**
  Tienen un significado especial para la shell y son uno de los siguientes caracteres:
  ; & ( ) | > > <espacio> <tab>
- **Operadores de control:**
  || & && ; ;; ( ) | <nueva-linea>
- **argumentos:**
  Son literales tomados como parámetros de entrada para algún comando.
- **comentarios:**
  Todo lo que sigue al carácter '#' hasta <nueva-linea> será un comentario.
- **Palbras reservadas:**
  Son palabras reservadas para implementar el lenguaje shell-script. Son palabras reservadas:
  case, do, done, elif, else, esac, fi, for, function, if, in, select, then, until, while, time.
- **Comandos internos:**
  Comandos que están implementados dentro de la propia shell. No necesitan PATH. Ejemplos de comandos internos son:
  cd, exec, arg, eval, exit,...

Para una referencia exacta y completa debería acudir a la página man bash(1) aunque eso resultaría excesivo en este momento.

Vamos a dar una serie de ejemplos para que los practique en su ordenador. No es necesario que introduzca los comentarios.

```
$ # Orden con un único comando sin opciones ni argumentos
$ ls

$ # Orden sin ociones pero con tres argumentos
$ ls . / ..

$ # Orden con un comando y una opción
$ ls -l

$ # Orden con un comando y tres opciones indicada de varias formas
$ # distintas pero equivalentes entre si.
$ ls -trl
$ ls -rtl
$ ls -ltr
$ ls -l -t -r
$ ls     -l    -t     -r
```

```
$ ls -lt -r
$ ls -l -tr

$ # Opciones que empiezan con '--'
$ ls --help
$ ls --version
$ ls --color=auto

$ # Ejemplo de opciones y argumentos sensibles al orden.
$ date -d now -R
$ date -d -R now

$ # ejemplo de opcion que empieza por '+'
$ date +'%a %b %e %H:%M:%S %Z %Y'
```

Expansión de la linea de orden:
Existe un detalle muy importante en el funcionamiento de la shell. Una cosa es lo que nosotros escribimos y otra lo que la shell ordena que se ejecute. Antes de que la shell ejecute comando alguno expande la linea de ordenes. Es decir esa linea se transforma en otra linea más larga. La orden resultante puede ser muy larga y tener muchos argumentos. Por ejemplo un '*' será sustituido por la lista de ficheros que se encuentren en el directorio actual. Quizas alguien encuentre que existe un parecido con el uso de '*.*' en Msdos por ejemplo pero el parecido es muy superficial y engañoso. Cuando en Unix hacemos 'ls *' el intreprete de comandos expande el asterisco y el comando ls recibe una lista de ficheros que tiene que listar. En Msdos cuando hacemos 'dir *.*' el interprete de comandos no expande nada. El comando dir recibe como argumento no la lista de ficheros que tiene que listar sino un '*.*' y será el propio comando dir quien tenga que expandir el argumento para obtener la lista de ficheros que tiene que listar. Por lo tanto la expansión en Unix está centralizada en el interprete de comandos el cual permite hacer expansiones mucho más potentes que en Msdos. Un comando no es más que un ejecutable y los ejecutables generalmente admiten argumentos. En Unix los comandos toman los argumentos que les pasa la shell después de la expansión.

El '*' se expande en base a los nombres de ficheros presentes en nuestro directorio actual, sustituyéndose por una cadena de caracteres cualquiera que no empieze por un punto.

Vamos a realizar una práctica completa. Algunos de los comandos que vamos a usar como 'cd', 'mkdir', 'touch' y otros son comandos que no explicamos en este momento pero que servirán para situarnos en un directorio de trabajo y crear algunos ficheros para la práctica.

Las lineas que empiecen con	$ comando ...	son comandos que debe introducir salvo que se trate de un comentario.
Las lineas en amarillo	salida del comando ...	son la salida obtenida.

Esto no quiere decir que siempre que pongamos un comando vayamos a poner a continuación su salida en amarillo pero si la ponemos es para que compruebe lo que debe obtener.

Ahora introduzca los comandos que indicamos a continuación y compruebe la salida obtenida.

```
$ cd /tmp
$ mkdir pruebas
$ cd pruebas
$ # Ya hemos creado un directorio de pruebas y ya estamos dentro de él.
$ # Para comprobarlo hacemos
$ pwd
/tmp/pruebas
$ # Ahora creamos unos ficheros para practicar
$ touch kk1 kk2 kkkk kk.txt kk.doc j2.txt .kk
$ echo *
kk1 kk2 kkkk kk.txt kk.doc j2.txt
$ echo k*
kk1 kk2 kkkk kk.txt kk.dox
$ echo *xt
kk.txt j2.txt
$ echo *.

$ echo .*
.kk
$ echo *.*
kk.txt kk.doc j2.txt
```

Fijese que los resultados dependen de los ficheros existentes en el directorio actual. Los mismos comandos realizados desde otro directorio distinto darían otro resultado.

Mantenga la sesión de la práctica en este punto porque continuaremos haciendo algunas prácticas más desde este mismo punto.

El '?' se expande como un único carácter y tampoco expande un punto en el comienzo del nombre del fichero. Introduzca ahora los siguientes comandos y compruebe la salida obtenida.

```
$ echo ???
kk1 kk2
$ echo kk?
kk1 kk2
```

A continuación no teclee nada ya que el resultado es solo una hipotesis. Supongamos que obtenemos un error de este tipo.

```
$ ls *
ksh: /bin/ls: arg list too long
```

Esto significa que el '*' se expande en un número demasiado grande de ficheros y eso resulta un problema para el interprete de comandos. La forma de obtener la lista de ficheros sería haciendo

```
$ ls
```

o también.

```
$ ls .
```

Si quisieramos averiguar el número de ficheros podríamos contarlos con 'wc'. Aqui estamos usando un '|' que aun no hemos explicado pero que lo explicaremos en el próximo capítulo de redirección de entrada salida.

```
$ ls | wc -l
```

Peculiaridades de expansión en la bash de Linux
Solo vamos a mencionar unas cuantas y remitimos al lector a la página del manual de bash para más informacion.

- Expansión de corchetes.
  Este es un tipo de expansión que no tiene para nada en cuenta los ficheros existentes en el directorio actual.

```
$ echo a{d,c,b}e
ade ace abe
```

- Expansión de la tilde.
  Esta es una característica especial de la shell de Linux que resulta realmente util. Expande la tilde como directorio home.

```
$ echo ~
$ echo ~root
$ echo ~root/*
```

- Arithmetic Expansion.
  Esto permite expandir expresiones $((expression))

```
$ echo $(((4+11)/3))
5
```

En bash(1) en el apartado de EXPANSION se mencionan más cosas.

Redirección de entrada salida:
Normalmente los proceso utilizan para entrada y salida de datos unos dispositivos estandar. Son entrada estandar, salida estandar y salida estandar de errores. Generalmente se utiliza como entrada estandar la entrada de teclado y como salida estandar y salida estandar de errores la pantalla. La salida estandar se usa como salida normal de datos y la salida estandar de errores se usa para sacar mensajes que indican algún error, aunque

también se usa simplemente como flujo alternativo en caso de que no resulte deseable su mezcla con otros datos que deben salir por salida estandar.

Se puede alterar flujo de datos que va desde un dispositovo estandar a un programa o viceversa puede ser redirigido a otro dispositivo, o a otro programa, o fichero, etc..

Desde la shell se puede hacer esto de varias formas.

- >
  Redirige la salida estándar a un fichero o dispositivo.
- <
  Redirige la entrada estándar tomándola desde un fichero.
- |
  Comunica dos procesos por medio de entrada salida. Ojo no confundir con Msdos. En Unix los procesos comunican directamente pasándose los datos directamente sin crear ningún fichero temporal. El proceso que lee quedara en espera mientras el el proceso que escribe mantenga abierto el dispositivo de salida estándar incluso si momentáneamente no se produce salida. Cuando el proceso escritor termina cierra todos sus ficheros y el proceso lector acusará la condición como un fin de datos de entrada.
- >>
  Redirige la salida estándar a un fichero sin sobreescribirlo. En lugar de ello añade al final del mismo.
- <<FIN
  Redirige entrada estándar desde la propia linea de ordenes. (En lugar de FIN se puede usar cualquier literal).
- 2>
  Redirige la salida estándar de errores a un fichero o dispositivo.
- 2>&1
  Redirige la salida estandard de errores donde esta redirigido la salida estandard. (0=entrada estandar, 1=salida estándar, 2=salida de errores estándar)

Ejemplos

La orden siguiente no produce ningún resultado visible porque la salida estándar se redirige al dispositivo /dev/null. Este dispositivo es como un pozo sin fondo. A diferencia de una papelera de Windows no se puede recuperar luego nada.

```
$ date > /dev/null
```

Ahora un ejemplo curioso. El comando 'time' sirve para medir consumos de tiempos de otros comandos y para evitar mezclas de datos en salida estandar se decidió que la salida normal de time fuera la salida estandar de errores.

```
$ time whoami > /dev/null
```

Podemos ver el consumo de tiempo del comando 'whoami' porque este sale por la salida estandar de errores. Es un ejemplo de utilización de la salida estandar de errores para sacar información que no tiene nada que ver con

ningún error. Sin embargo time es un comando interno y por ello lo siguiente no funcionaría como usted piensa.

```
$ time whoami 2> /dev/null
```

En este caso la redirección afecta solo al comando whoami.

Los comandos internos son parte de la shell y para redirigir su salida habría que redirigir la salida completa de la shell. Dejaremos esto para un capítulo posterior.

Antes de continuar vamos a asegurarnos que estamos en un sitio seguro para trabajar.

```
$ cd /tmp
$ mkdir pruebas > /dev/null
$ cd pruebas
$ # Para comprobar que estamos en '/tmp/pruebas' hacemos
$ pwd
/tmp/pruebas
```

El contenido de '/tmp' suele ser vaciado cuando rearranca la máquina o quizas en algún otro momento. Contiene información temporal. Hemos usado /dev/null para ignorar la salida de errores del comando mkdir. Si ya existía '/tmp/pruebas' mkdir habría dado un error pero nosotros lo ignoramos porque solo nos intersa que lo cree en caso de que no exista y en caso contrario da igual. El dispositivo '/dev/null' también resulta útil para simular una entrada nula de datos. Por ejemplo para crear un fichero vació. Si ya estamos situados en '/tmp/pruebas' pruebe lo siguiente:

```
$ cat < /dev/null > kk
$ ls -l kk
```

El mismo efecto podríamos conseguir usando.

```
$ > kk
$ ls -l kk
```

Esto se puede utilizar para vaciar ficheros respetando los permisos originales.

Vamos mirar el contenido del directorio raiz

```
$ ls -l
```

Ahora queremos repetir el comando pero guardando el resultado en el fichero kk

```
$ ls -l / > kk
```

No vemos nada porque toda la salida va a parar al fichero. Para visualizar el contenido de un fichero usaremos el comando 'cat'.

```
$ # Mostrar el contenido del fichero kk.
$ cat kk

$ # Equivale al anterior.
$ cat < kk

$ # Parecido a los dos anteriores. /dev/tty es un dispositivo que
$ # se identifica como nuestro terminal.
$ cat < kk > /dev/tty
```

También podemos visualizar la salida de un comando a la vez que guardamos la salida en un fichero. El comando tee actua como un bifurcación.

```
$ ls -l / | tee kk1
$ cat kk1
```

Recuerde que lo que sigue a un '|' ha de ser siempre un ejecutable.

```
$ touch kk
$ date | kk
```

Esto habrá dado un error. Probemos ahora a redirigir la salida estandar de errores a un fichero y la salida estandar a otro.

```
S touch kk
$ date | kk 2> errores > salida
$ cat errores
$ cat salida
```

Existe un comando 'yes' que sirve para generar continuamente respuestas afirmativas en forma de caracteres 'y' y finales de linea. Este comando está pensado para ser usado con '|' y proporcionar entradas afirmativas a un comando. Usaremos el comando 'head' que sirve para sacar por salida estandar solo una primera parte de los datos de la entrada. La cantidad datos puede especificarse en número de lineas en bytes, etc.. Nosotros vamos a utilizar bytes. La salida de 'yes' la entregaremos a head para que tome los 1000 primeros bytes que pasamos al programas 'wc' para que cuente lineas palabras y caracteres.

```
$ yes | head --bytes=1000 | wc
```

```
500 500 1000
Tubería rota
```

El comando 'yes' podría funcionar eternamente. Su salida redirigida por las buenas a un fichero llenaría el disco (cosa nada desable por cierto) y acabaría dando un error. En lugar de esto lo hemos redirigido a un programa que aceptará solo los 1000 primeros caracteres y luego cerrará su entrada provocando un error de Tubería rota.

La shell también permite introducir datos en una forma especial que se llama documento-aqui. Para variar un poco usaremos ahora el comando 'sort' que sirve para ordenar. Observe que en este ejemplo un fin de linea no termina el comando. Por ello aparecerá el introductor secundario $PS2 que nosotros indicamos con un '>' en amarillo.

```
$ # Para ordenar unas lineas que introducimos en la propia linea de
$ # ordenes usamos el operador '<<' seguido de una clave de
$ # finalización, de la entrada.
$ sort <<FIN
> aaaaa
> cccccc
> zzzzz
> bbbb
> yyyy
> FIN

aaaaa
bbbb
cccccc
yyyy
zzzzz
```

Operador grave:
Cuando colocamos algo dentro de las comillas graves '`' la shell lo interpretara como una cadena cuyo valor es sustituido por el resultado de ese comando que produce ese comando en salida estándar.

```
$ echo date produce --- `date` ---

date produce --- lun ene 10 20:59:12 CET 2000 ---
```

El resultado es solo un ejemplo y no puede coincidir exactamente con el resultado obtenido por usted ya que debería poner la fecha correcta en su sistema.

Quizas el ejemplo que acabamos de usar no le de una idea exacta de la potencia del operador grave. Recuerde que esto ocurre durante la expansión de la linea de ordenes antes de ejecutarse el comando echo. Por ello podemos hacer cosas como las siguientes.

```
$ # Ejecutamos date y guardamos el resultado en un fichero cuyo
$ # nombre construimos en base a nuestro nombre de usuario y al
$ # año de la fecha actual.
$ date > fichero-`whoami`-`date +%Y`
```

```
$ ls -l fichero-*
$ cat fichero-*
```

Hemos usado un comando para construir el nombre del fichero de salida. No mostramos los resultados pero confiamos en que ya los ha comprobado.

Caracteres de escape:
Dado que la Shell interpreta catarteres. Blancos como separadores. Asteriscos e interrogaciones como comodines. Operador grave. Comillas dobles, Comillas normales, $, etc... Se plantea el problema de que hacer cuando queremos usar estos caracteres como tales sin que sean interpretados por la shell. Existen varias formas de escapar caracteres para que la shell no los expanda, o interprete.

La shell no el el único programa con capacidad de expandir e interpretar caracteres especiales. Por ejemplo find, egrep, sed, y otros también interpretan ciertos caracteres que además pueden coincidir con algunos de los que interpreta la shell. Para usar caracteres especiales que pueden interpretarse por la shell habrá que escaparlos siempre que deseemos que lleguen al comando.

Vamos a ver tres formas distintas de escapar caracteres:

- Cuando usamos las dobles comillas, los comodines asterisco, interrogación y caracteres en blanco no se interpretan, aunque si se interpretan '$', comillas graves '`', y otros.
- Cuando usamos comillas simples se respeta todo el contenido de la cadena sin expandir nada.
- Cuando usamos el carácter '\' escapamos el carácter que va inmediatamente después. Cuando es un carácter que tiene un significado especial para la shell, dejará de tener ese significado especial. Vamos a ver todo esto con unos ejemplos:

```
$ # La orden siguiente muestra el mismo literal que el entre comillas.
$ echo "* * ? *"

$ # En este caso la shell interpreta `pwd` y $PATH, en cambio '*' y
$ # '?' no se interpretan.
$ echo "* ? `pwd` $PATH"

$ # En este caso se conserva todo el literal sin interpretar.
$ echo '* ? `pwd` $PATH'

$ # A continuación la orden mostrará dos comillas dobles.
$ echo \"\"

$ # El caracter <nueva-linea> también puede ser escapado y en este
$ # caso puede servir para introducir comandos muy largos de forma
$ # más legible. Para hacer esta prueba deberá pulsar la tecla de
$ # <nueva-linea> justo a continuación del caratcter '\'
$ ls -l / | \
$ head -n 10 | \
$ tee /tmp/resultado | \
$ wc
```

Ejercicio:
Las siguientes ordenes producen resultados cuya explicación no resulta trivial. Esto es debido a que permitmos con la opción -e que el comando 'echo' interprete caracteres. Por lo tanto lo que ocurre es que primero el interprete de comandos interpreta los caracteres de la linea de órdenes y luego el comando echo interpreta los caracteres recibidos. Consulte la página man de echo(1) y busque la opción -e. Ejecute los comandos que le proponemos a continuación e intente explicar los resultados. Puede que necesite papel y lapiz.

```
$ echo -e \n
n
$ echo -e \\n


$ echo -e \\\n


$ echo -e \\\\n
\n $ echo -e \\\\\n
\n $ echo -e \\\\\\n
\
$ echo -e \\\\\\\n
\
$ echo -e \\\\\\\\n
\\n
```

Resumen
Hemos visto muchas cosas que nos permiten hacernos idea de la potencia de la Bourne-Shell de Unix. La bash de Linux es un superconjunto. Es decir es compatible pero más potente.

Queremos resaltar que para las personas que han conseguido terminar este capítulo ya se ha conseguido algo importante. Personas acostumbradas al usos de Msdos cometen barbaridades que luego ni siquiera son capaces de explicar. La clave muchas veces está en la expansión de la linea de ordenes. En realidad este tema no está agotado ni mucho menos y lo completaremos en capítulos posteriores aunque intercalaremos material más ligero porque no deseamos hacerle sufrir demasiado. Solo lo justo y por ello creo que es el momento de confesar que existía un truco para realizar más facilmente el ejercicio anterior aunque no lo mencionamos antes porque habría resultado demasiado fácil y cómodo. Se trata de activar la traza de la shell con 'set -x'. De esta forma podemos ver el resultado de la expansión de linea de órdenes precedida por el signo '+'.

```
$ set -x
+ set -x
$ echo -e \\\\\\\n
+ echo -e '\\\\n'
\\n
$ set +x
+ set +x
```

Puede utilizar este truco cuando no sepa que está ocurriendo con la expansión de una linea de ordenes que no funciona como usted pensaba.

INTRODUCCION A LOS PROCESOS

- Que es un proceso
- PID y PPID
- El proceso init
- UID y EUID
- Como se crea un nuevo proceso
- La llamada al sistema exec()
- La llamada al sistema fork()
- Test

Que es un proceso
A estas alturas más de uno empezará a preguntarse si realmente es necesario conocer todos estos detalles para un simple uso de un SO. Este curso esta orientado al uso de Linux desde la consola y resulta necesario conocer estas cosas incluso para un usuario normalito. No vamos a describir detalladamente como están implementados los procesos en Linux.

Quizas para un curso de programación avanzada si fuera necesario, pero lo que nosotros vamos a describir es unicamente los detalles más importantes desde un punto de vista didáctico y práctico para un usuario normal.

En un sistema multitarea todo funciona con procesos así que conocer unos cuantos principios básicos sobre procesos le resultará de la máxima utilidad. En un sistema monotarea se usa frecuentemente el término programa indistintamente para hablar de un programa en papel, en cdrom, en disco duro o en funcionamiento. En un sistema multitarea no se debe usar el término programa para hablar de la ejecución del mismo. En su lugar hablaremos de proceso indicando con ello que esta arrancado y funcionando. Un programa puede dar lugar a varios procesos. Por ejemplo en un mismo instante varios usuarios pueden estar usando un mismo editor. Un proceso puede estar detenido pero a diferencia de un programa existe una informacíon de estado asociada al proceso. Un programa es algo totalmente muerto. Un proceso detenido es más bien como si estuviera dormido.

Los procesos tienen organizada la memoria de una forma especial muy eficiente. Por ejemplo la párte de código del proceso es una parte de solo lectura y puede ser compartida por varios procesos a la vez. Imaginemos que hay varios usuarios en el sistema usando un mismo editor. En ese caso sería un desperdicio tenener la mísma información de código de ese programa repetida varias veces en memoria y ocupando un recurso tan valioso como es la memoria RAM.

En el caso de tener varios programas distintos en ejecución también se suele dar el caso de que contengan pártes comunes de código pertenecientes a librerías que continen gran cantidad de funciones de propósito general. Para hacer un uso eficiente de estas librerías existen librerías dinámicas de uso compartido. En Linux el uso de estas librerías no está organizado de una forma unificada para todas las distribuciones por lo cual los ejecutables binarios pueden ser incompatibles entre distintas distribuciones. Este problema se puede solucionar partiendo de los fuentes y recompilando la aplicación en nuestro sistema. Por esta razón un binario de RedHat, o Suse puede no funcionar en Slackware o en Debian.

PID y PPID
A cada proceso le corresponderá un número PID que le identifica totalmente. Es decir en un mismo momento es imposible que existan dos procesos con el mismo PID. Lo mismo que todos los procesos tienen un atributo PID que es el número de proceso que lo identifica en el sistema tambien existe un atributo llamado PPID. Este número se corresponde con el número PID del proceso padre. Todos los procesos deben de tener un proceso

que figure como padre pero entonces que ocurre si un padre muere antes que alguno de sus hijos ? En estos casos el proceso 'init' del cual hablaremos en seguida adoptará a estos procesos para que no queden huerfanos.

El proceso init

Cuando arranca el sistema se desencadena una secuencia de procesos que a grandes rasgos es la siguiente. Primero se carga el nucleo de Linux (Kernel) de una forma totalmente especial y distinta a otros procesos. Dependiendo de los sitemas puede existir un proceso con PID=0 planificador, o swapper. En Linux y en casi todos los sistemas tipo Unix seguirá un proceso llamado 'init'. El proceso init tiene PID = 1. Lee un fichero llamado inittab donde se relacionan una serie de procesos que deben arrancarse para permanecer activos todo el rato (demonios). Algunos de ellos están definidos para que en caso de morir sean arrancados de nuevo inmediatamente garantizando la existencia de ese servicio de forma permanente.

Es decir 'init' es un proceso que va a generar nuevos procesos pero esta no es una cualidad especial. Es muy frecuente que un proceso cualquiera genere nuevos procesos y cuando esto ocurre se dice que genera procesos hijos.

Este no es un curso de administración pero diremos que a init se le puede indicar que arranque el sistema de diferentes formas, por ejemplo en modo monousuario para mantenimiento. Este es un capítulo en el cual pueden surgir muchas preguntas retorcidas tales como, que pasa si matamos a init, o quien es el padre de init, pero no nos interesa responder a esto ya que init es un proceso muy especial y su padre aún más. En cada sistema de tipo Unix las respuestas a cosas como estas pueden variar mucho porque a ese nivel la implementaciones varían mucho. Ningún programa normal necesitará usar ese tipo de información. Quedan muchos detalles interesantes relativos a temas de administración. Los curiosos siempre tienen el recurso de mirar la página man de init(8) y de inittab(5) pero nosotros no insistiremos más en este tema. (Que alivio verdad ?)

UID y EUID

Los procesos tienen un EUID (Efectiv User Identif), y un UID normalmente ambos coinciden. El UID es el identificador de usuario real que coincide con el identificador del usuario que arrancó el proceso. El EUID es el identificador de usuario efectivo y se llama así porque es el identinficador que se tiene en cuenta a la hora de considerar los permisos que luego explicaremos.

El UID es uno de los atributos de un proceso que indica por decirlo de alguna manera quien es el propietario actual de ese proceso y en función de ello podrá hacer ciertas cosas. Por ejemplo si un usuario normal intentara eliminar un proceso del cual no es propietario el sistema no lo permitirá mostrando un mensaje de error en el que advierta que usted no es el propietario de ese proceso y por tanto no está autorizado a hacer esa operación. Por el contrario el usuario root puede hacer lo que quiera con cualquier proceso ya que el sistema no comprueba jamás si root tiene permisos o no para hacer algo. root siempre tiene permisos para todo. Esto es cierto a nivel de llamadas del sistema pero nada impide implementar un comando que haga comprobaciones de permisos incluso con root. Algunos comandos en Linux tienen opciones que permiten hacer estas cosas y solicitar confirmación en caso de detectar una operación peligrosa o infrecuente.

El UID también es un atributo presente en otros elementos del sistema. Por ejemplo los ficheros y directorios del sistema tienen este atributo. De esta forma cuando un proceso intenta efectuar una operación sobre un fichero. El kernel comprobará si el EUID del proceso coincide con el UID del fichero. Por ejemplo si se establece que determinado fichero solo puede ser leido por su propietario el kernel denegará todo intento de lectura a un proceso que no tenga un EUID igual al UID del fichero salvo que se trate del todo poderoso root.

Aunque estamos adelantando cosas sobre el sistema de ficheros vamos a continuar con un ejemplo. Comprobaremos cual es el UID de su directorio home.

```
$ # Cambiamos el directorio actual a home
$ cd
$ # Comprobamos a quien pertenece 'uid' y 'guid'
$ ls -ld .
$ # Ahora obtenemos el 'uid' y el 'gid' con sus valores numéricos.
$ ls -lnd .
```

Si su directorio home está configurado de forma lógica deberá pertenecerle a usted. Si esto no es así reclame enérgicamente a su administrador, pero si el administrador resulta ser usted sea más indulgente y limitese a corregirlo y no confiese su error a nadie. En realidad casi todo lo que que se encuentre dentro de su directorio home debería pertenecerle a usted. Usted debe ser el propietario de su directorio home porque de otra forma y dependiendo de los permisos asociados a este directorio los procesos que usted arranque se verían o bien incapaces de trabajar con él, o lo que es peor cualquiera podría hacer cualquier cosa con él. Como es lógico hemos mencionado de pasada el tema de permisos de directorios para ilustrar un poco la utilidad del uid pero esto se verá en detalle en el capítulo dedicado al sistema de ficheros de Linux.

Algunos ficheros ejecutables poseen un bit de permisos que hace que cambie el EUID del proceso que lo ejecute convirtiendose en el UID del propietario del fichero ejecutado. Suena complicado pero no lo es. Es decir imaginemos que usted ejecuta un comando propiedad de root que tiene este bit. Pues bien en ese momento el EUID de su proceso pasaría a ser el de root. Gracias a esto un proceso puede tomar temporalmente la identidad de otro usuario. Por ejemplo puede tomar temporalmente la identidad de root para adquirir privilegios de superusuario y así acceder por ejemplo a ficheros del sistema propiedad de root.

El sistema recibe continuamente peticiones de los procesos y el EUID del proceso determinará que el kernel le conceda permiso para efectuar la operación deseada o no.

Muchas veces sorprende que en Linux apenas se conozcan unos pocos casos de virus, mientras que en otros sistemas parecen estar a la orden del día. Es perfectamente posible realizar un virus que infecte un sistema Linux pero de una forma o de otra el administrador tendría que darle los privilegios que le convierten en peligroso. Por eso no es facil que estos virus lleguen muy lejos.

Como se crea un nuevo proceso

El núcleo del sistema llamado también kernel es el encargado de realizar la mayoría de funciones básicas que gestiona entre otras cosas los procesos. La gestíon de estas cosas se hacen por medio de un limitado número de funciones que se denominan llamadas al sistema. Estas llamadas al sistema están implementadas en lenguaje C y hablaremos ahora un poco sobre un par de ellas llamadas fork() y exec(). Si logramos que tenga una vaga idea de como funcionan estas dos importantísimas funciones facilitaremos la compresión de muchas otras cosas más adelante.

La llamada al sistema exec()

Cuando hablamos de proceso debe usted pensar solamente en algo que se está ejecutando y que está vivo. Un proceso puede evolucionar y cambiar totalmente desde que arranca hasta que muere. Lo único que no cambia en un proceso desde que nace hasta que se muere es su identificador de proceso PID. Una de las cosas que puede hacer un proceso es cambiar por completo su código. Por ejemplo un proceso encargado de procesar la entrada y salida de un términal (getty) puede transformarse en un proceso de autentificacion de usuario y password (login) y este a su vez puede transformarse en un interprete de comandos (bash). Si la llamada exec() falla retornará un -1. Esto no es curso de programación así que nos da igual el valor retornado pero lo que si nos interesa es saber que cuando esta llamada tiene éxito no se produce jamás un retorno. En realidad no tendría sentido retornar a ningún lado. Cuando un proceso termina simplemente deja de existir. En una palabra

muere y ya está. La llamada exec() mantiene el mismo PID y PPID pero transforma totalmente el código de un proceso en otro que cargará desde un archivo ejecutable.

La llamada al sistema fork()
La forma en que un proceso arranca a otro es mediante una llamada al sistema con la función fork(). Lo normal es que el proceso hijo ejecute luego una llamada al sistema exec(). fork() duplica un proceso generando dos procesos casi idénticos. En realidad solo se diferenciaran en los valores PID y PPID. Un proceso puede pasar al proceso hijo una serie de variables pero un hijo no puede pasar nada a su padre a través de variables. Además fork() retorna un valor numérico que será -1 en caso de fallo, pero si tiene éxito se habrá producido la duplicación de procesos y retornará un valor distinto para el proceso hijo que para el proceso padre. Al proceso hijo le retornará el valor 0 y al proceso padre le retornará el PID del proceso hijo. Después de hacer fork() se pueden hacer varias cosas pero lo primero que se utiliza despúes del fork es una pregunta sobre el valor retornado por fork() para así saber si ese proceso es el padre o el hijo ya que cada uno de ellos normalmente deberá hacer cosas distintas. Es decir la pregunta sería del tipo si soy el padre hago esto y si soy el hijo hago esto otro. Con frecuencia aunque no siempre el hijo hace un exec() para trasformarse completamente y con frecuencia aunque no siempre el padre decide esperar a que termine el hijo.

También normalmente aunque no siempre esta parte de la lección es necesario releerla más de una vez.

Estamos dando pequeños detalles de programación porque en estas dos llamadas del sistema son muy significativas. Su funcionamiento resulta chocante y su comprensión permite explicar un montón de cosas.

MAS SOBRE PROCESOS Y SEÑALES

- Las formas de comunicación entre procesos
- Secuencia de arranque en una sesión de consola
- Comando ps
- Configuración del terminal
- Comando time
- Comando kill
- Comando nice
- Comando renice
- Test

Las formas de comunicación entre procesos
Los procesos no tiene una facilidad de acceso indiscriminada a otros procesos. El hecho de que un proceso pueda influir de alguna manera en otro es algo que tiene que estar perfectamente controlado por motivos de seguridad. Comentaremos solo muy por encima las diferentes formas de comunicación entre procesos.

1. A través de variables de entorno:
   Solo es posible de padres a hijos.
2. Mediante una señal:
   Solo indica que algo ha ocurrido y solo lleva como información de un número de señal.
3. Mediante entrada salida:
   Es la forma más corriente a nivel de shell. Ya hemos comentado el operador pipe '|' que conecta dos procesos.
4. Mediante tégnicas IPC u otras:
   Semáforos, Memoria compartida, Colas de mensages.

5. Mediante sockets:
   Este sistema tiene la peculiaridad de que permite comunicar procesos que estén funcionando en máquinas distintas.

No profundizamos sobre esto porque ahora no estamos interesados en la programación. Más adelente si comentaremos bastante sobre variables y entrada salida porque daremos nociones de programación en shell-script. Tambien daremos a continuación unos pequeños detalles que tienen que ver con el arranque del sistema porque también nos resultará util para comprender como funcionan las cosas.

Secuencia de arranque en una sesión de consola

Para consultar la dependencia jerárquica entre unos procesos y otros existe en Linux el utilísimo comando pstree. No es esencial y quizas no lo tenga usted instalado pero resulta muy práctico. Si dispone de él pruebe los comandos 'pstree', y 'pstree -p'. Nosotros vamos a mostrar el resultado de ambos comandos en el sistema que estamos usando en este momento para que en cualquier caso pueda apreciar el resultado pero le recomendamos que lo instale si no dispone de él ya que resulta muy práctico. Tambien puede usar como sustituto de 'pstree' el comando 'ps axf' pero no espere un resultado tan bonito.

```
$ pstree

init-+-apache---7*[apache]
     |-atd
     |-bash---pstree
     |-2*[bash]
     |-bash---vi
     |-bash---xinit-+-XF86_S3V
     |              `-mwm-+-.xinitrc---xterm---bash
     |                    `-.xinitrc---xclock
     |-cron
     |-getty
     |-gpm
     |-inetd
     |-kflushd
     |-klogd
     |-kpiod
     |-kswapd
     |-kupdate
     |-lpd
     |-portmap
     |-postmaster
     |-sendmail
     |-sshd
     |-syslogd
     `-xfs
```

Este formato nos da un arbol de procesos abreviado en el que procesos con el mismo nombre y que tengan un mismo padre aparecerán de forma abreviada. En el ejemplo anterior aparece '2*[bash]' indicando que hay dos procesos bash como hijos de init, y tambien hay algunos proceso apache arrancados. El que existan muchos procesos arrancados no indica necesariamente un alto consumo de CPU. Puede que estén todos ellos haciendo el vago. Bueno en el caso de apache quizas estén haciendo el indio. (Esto último es una broma que no he podido evitar).

```
$ pstree -p

init(1)-+-apache(204)-+-apache(216)
        |             |-apache(217)
        |             |-apache(218)
        |             |-apache(219)
        |             |-apache(220)
        |             |-apache(1680)
        |             `-apache(1682)
        |-atd(196)
        |-bash(210)---pstree(1779)
        |-bash(211)
        |-bash(212)---vi(1695)
        |-bash(215)---xinit(1639)-+-XF86_S3V(1644)
        |                         `-mwm(1647)-+-.xinitrc(1652)---xterm(1660)---bash(1661)
        |                                     `-.xinitrc(1655)---xclock(1673)
        |-bash(214)
        |-cron(199)
        |-getty(213)
        |-gpm(143)
        |-inetd(138)
        |-kflushd(2)
        |-klogd(131)
        |-kpiod(4)
        |-kswapd(5)
        |-kupdate(3)
        |-lpd(153)
        |-portmap(136)
        |-postmaster(168)
        |-sendmail(179)
        |-sshd(183)
        |-syslogd(129)
        `-xfs(186)
```

En este otro formato. Aparece cada proceso con su PID. Podemos ver que el Proceso 'init' tiene pid = 1 y ha realizado varios forks() generando procesos con pid > 1. En algunos sistemas la generación de números de PID para procesos nuevos se realiza en secuencia. En otros resulta un número impredecible.

Entre los procesos generados por 'init' están los procesos 'getty'. Se arrancará un 'getty' por cada terminal. Este proceso configura la velocidad y otras cosas del terminal, manda un saludo y luego se transforma con exec el proceso 'login'. Todos estos procesos se ejecutan con EUID y UID = 0, es decir como procesos del superusuario root. Cuando el proceso 'login' conoce nuestra identidad después de validar usuario password se transformará con exec en la shell especificada e para nuestro usuario el fichero /etc/passwd

Para ver la linea que contiene sus datos pruebe a hacer lo siguiente:

```
$ grep `whoami` /etc/passwd
```

La linea tiene el siguiente formato. login:contraseña:UID:GID:nombre:dir:intérprete

Vamos a suponer que su shell por defecto sea la bash. Si esta shell arrancara con el EUID = 0 tendríamos todos los privilegios del super usuario pero esto no ocurre así. Esta shell ya tendrá nuestro UID y nuestro EUID. Vamos a representar todo esto marcando los puntos en los que ocurre algún fork() con un signo '+'.

```
[init]-+fork()->[getty]
       |
       +fork()->[getty]-exec()->[login]-exec()->[bash]+fork()-exec()->[comando]
       |
       +fork()->[getty]
       |
```

La shell puede arrancar un comando mediante un fork() y luego un exec() y esperar a que este muera. Recuerde que la función exec() no tiene retorno posible ya que finaliza con la muerte del proceso. En ese momento la shell detecta la muerte de su hijo y continua su ejecución solicitando la entrada de un nuevo comando. Cuando introducimos el comando 'exit' estamos indicando a la shell que finalice y su padre 'init' se encargará de lanzar nuevo proceso 'getty'. Lógicamente 'exit' es un comando interno de la shell. Quizas le llame la atención que la muerte de 'bash' termine provocando un nuevo 'getty' cuando 'getty' pasó a 'login' y este a 'bash' pero en esta secuencia getty-login-bash no hay ningún fork() por eso getty, login, y bash son en realidad el mismo proceso en distintos momentos con el mismo PID obtenido en el fork() realizado por 'init' solo que ha ido cambiando su personalidad manteniendo la misma identidad (mismo PID). Para 'init' siempre se trató del mismo hijo y la muerte de cualquiera de ellos (getty, login o bash) provoca que se arranque un nuevo 'getty' sobre ese mismo terminal con el fin de que ese terminal no quede sin servicio.

La presentación del mensaje de Login es mostrada por 'getty'. Una vez introducido el identificador de usuario será 'login' quien muestre la solicitud de introducción de la password, y una vez introducido el password será la shell quien muestre el introductor de comandos pero estamos hablando siempre del mismo proceso.

A modo de ejercicio compruebe estas cosas por usted mismo usando algunos de los comandos que ya conoce. Para hacer esta práctica no basta con usar un terminal remoto sino que necesitará un PC completo para ir haciendo cosas desde distintas sesiones. Le proponemos hacerlo más o menos de la siguiente manera:

1. Entre en cada uno de los terminales disponibles de forma que todos los terminales esten ocupados por un interprete de comandos. Bastará con hacer login en todos ellos.
2. Luego compruebe que no hay ningún proceso 'getty'.
3. Haga un exit desde uno de estos terminales.
4. Compruebe desde otro termina que ahora si existe un proceso 'getty' y anote su pid.
5. Introduzca el nombre de usuario en ese terminal que quedó libre.
6. Compruebe ahora desde otra sesion que existe un proceso login con el PID que usted anotó.
7. Termine de indentificarse tecleando la password
8. Compruebe desde otra sesión que ahora existe una shell con el PID que anotamos.

   Si no tiene el comando 'pstree' tendrá que usar 'ps' pero con pstree puede ver más facilmente lo que ocurrirá ahora.

9. Ahora teclee el comando 'sleep 222' desde la sesión que tiene el PID anotado por usted.
10. Compruebe desde otra sesión que el interprete de comandos ha realizado un fork() generando un comando que está ejecutando el comando indicado.

Si no ha podido realizar el ejercicio anterior tendrá que confiar en que las cosas son como decimos y ya está.

Comando ps
Muestra los procesos activos. Este comando es muy util para saber que comandos están funcionando en un determinado momento.

Siempre que se mencione un comando puede consultar la página man del mismo. En el caso de 'ps' se lo recomendamos ya que es un comando muy util con una gran cantidad de opciones. Nostros mencionaremos algunos ejemplos pero se aconseja probar 'ps' probando las distintas opciones que se mencionan en la página del manual.

Ejemplos:

```
$ # Para ver todos sus procesos que están
$ # asociados a algún terminal.
$ ps

PID TTY STAT TIME COMMAND
.......


$ # Para ver todos sus procesos y los de otros
$ # usuarios siempre asociados a algún terminal.
$ ps a

PID TTY STAT TIME COMMAND
.......


$ # Para ver todos sus procesos estén asociados o
$ # no a algún terminal.
$ ps x

PID TTY STAT TIME COMMAND
.......


$ # Para ver todos los proceso asociados al
$ # terminal 1
$ ps t1

PID TTY STAT TIME COMMAND
.......


$ # Para ver todos los procesos del sistema.
$ ps ax

PID TTY STAT TIME COMMAND
.......
```

Estos ejemplos que acabamos de ver obtienen un mismo formato de datos. Explicaremos el significado de estos atributos.

PID	Es el valor númerico que idenfica al proceso.
TTY	Es el terminal asociado a ese proceso. Los demonios del sistema no tienen ningún terminal asociado y en este campo figurará un ?
STAT	Tiene tres campos que indican el estado del proceso (R,S,D,T,Z) (W) (N) La S indica que el proceso está suspendido esperando la liberación de un recurso (CPU, Entrada Salida, etc) necesario para continuar. Explicaremos solo algunos de estos estados en su momento.
TIME	Indica el tiempo de CPU que lleva consumido ese proceso desde que fué arrancado.
COMMAND	Muestra el comando y los argumentos que le fueron comunicados.

Existen muchas opciones para el comando ps que ofrecen un formato distinto. Le recomendamos especialmente que pruebe 'ps u', 'ps l', y 'ps f'

En Unix los comandos suelen servir para una sola cosa, aunque suelen tener muchas opciones. La entrada de los comandos suele tener una estructura simple y la salida de los comandos tambíen. Si un comando no encuentra nada que hacer existe la costumbre de que termine de modo silencioso. Todo esto permite que los comandos puedan combinarse enganchado la salida de uno con la entrada de otro. Algunos comandos están especialmente diseñados para ser usados de esta forma y se les suele denominar filtros.

La salida del comando 'ps' se puede filtrar con 'grep' para que muestre solo las líneas que nos interesan.

Configuración del terminal

Conviene que comprobemos si su terminal está correctamente configurado para poder interrumpir un proceso. Normalmente se usa <Ctrl-C> pero esto depende de la configuración de su terminal. Si en algún momento su terminal queda desconfigurado haciendo cosas raras como por ejemplo mostrar caracteres extraños intente recuperar la situación tecleando el comando 'reset'. Esto solo es válido para Linux. Para otros sistemas puede ser util 'stty sane' que también funciona en Linux pero no es tan eficaz como el comando 'reset'. Para comprobar la configuración de su terminal puede hacer 'stty -a' aunque obtendrá demasiada información que no es capaz de interpretar, podemos indicarle que se fije en el valor de 'intr'. Debería venir como 'intr = ^C'. Si no lo localiza haga 'stty -a | grep intr'. De esta forma solo obtendrá una linea. Para configurar el terminal de forma que pueda interrumpir procesos con <Ctrl-C> puede intentar configurar su terminal haciendo 'stty ^V^C'. El carácter <Ctrl-V> no se mostrará en el terminal ya que actua evitando que el siguiente carater (<Ctrl-C> en nuestro caso) no sea interpretado como caracter de control.

No pretendemos ahora exiplicar los términales de Linux pero si queremos que compruebe su capacidad para interrumpir procesos con <Ctrl-C> ya que usaremos esto en las prácticas que siguen. Una prueba inofensiva para comprobar la interrupcion de un proceso es el siguiente comando que provoca una espera de un minuto. Deberá introducir el comando e interrumpirlo antes de que el tiempo establecido (60 segundos se agote).

```
$ sleep 60
<Ctrl-C>
```

Si no ha conseguido interrumpir el proceso no siga adelante para evitar que alguna de las prácticas deje un proceso demasiado tiempo consumiendo recursos de su máquina. Si esta usted solo en la máquina eso tampoco tendría mucha importancia pero es mejor que averigue la forma de interrumpir el proceso del ejemplo anterior.

Comando time
Da los tiempos de ejecucion. Este comando nos da tres valores cuya interpretacion es:

```
 real   Tiempo real gastado (Duración real)
user    Tiempo CPU de usuario.
sys.    Tiempo CPU consumido como proceso de kernel.
        (Es decir dentro de las llamadas al kernel)
```

La mayoría de los comandos están gran parte del tiempo sin consumir CPU porque necesitan esperar para hacer entrada salida sobre dispositivos lentos que además pueden estar en uso compartidos por otros procesos. Existe un comando capaz de esperar tiempo sin gastar tiempo de CPU. Se trata del comando 'sleep'. Para usarlo le pasaremos un argumento que indique el número de segundos de dicha espera.

Por ejemplo vamos a comprobar cuanta CPU consume una espera de 6 segundos usando sleep

```
$ time sleep 6

real    0m6.021s
user    0m0.020s
sys     0m0.000s
```

El resultado obtenido puede variar ligeramente en cada sistema pero básicamente obtendrá un tiempo 'real' de unos 6 segundos y un tiempo de CPU ('user' + 'sys') muy pequeño.

Vamos a medir tiempos en un comando que realice operaciones de entrada salida así como proceso de datos.

```
$ time ls /* > /dev/null


real    0m0.099s
user    0m0.080s
sys     0m0.010s
```

En este comando verá que el consumo total de CPU es superior al del comando sleep. En cualquier caso el tiempo real tardado en la ejecución del comando es siempre muy superior al consumo de CPU.

Vamos a probar un comando que apenas realice otra cosa que entrada salida. Vamos a enviar 10Mbytes al dispositivo /dev/null. Existe un comando 'yes' que provoca la salida continua de un caracter 'y' seguido de un caracter retorno de carro. Esta pensado para sustituir la entrada de un comando interactivo en el cual queremos contestar afirmativamente a todo lo que pregunte. Nosotros fitraremos la salida de 'yes' con el comando 'head' para obtener solo los 10Mbytes primeros producidos por 'yes' y los enviaremos al dispositivo nulo '/dev/null' que viene a ser un pozo sin fondo en el cual podemos introducir cualquier cosa sin que se llene, pero no podremos sacar absolutamente nada. En una palabra vamos a provocar proceso de entrada salida perfectamente inutil.

```
$ time yes | head --bytes=10000000 > /dev/null

Tubería rota

real    0m6.478s
user    0m5.440s
sys     0m0.900s
```

Podemos hacer un consumo fuerte de CPU si forzamos a cálculos masivos que no tengan apenas entrada salida. Por ejemplo podemos poner a calcular el número PI con 300 cifras decimales. 'bc' es un comando que consiste en una calculadora. Admite uso interactivo pero tambien acepta que le pasemos las operaciones desde otro proceso combinando entrada salida.

```
$ time ( echo "scale=300; 4*a(1)" | bc -l )

3.14159265358979323846264338327950288419716939937510582097494459230 7\
8164062862089986280348253421170679821480865132823066470938446095505 8\
2231725359408128481117450284102701938521105559644622948954930381964 4\
2881097566593344612847564823378678316527120190914564856692346034861 0\
4543266482133936072602491412 72

real    0m2.528s
user    0m2.520s
sys     0m0.010s
```

En un Pentium 200Mhz este comando tardó 3 segundos para 300 cifras y 20 segundos usando 600 cifras. Decimos esto para que vea que el tiempo que se tarda aumenta exponencialmente y que dependiendo de la potencia de su ordenador puede suponer bastante tiempo de proceso. Quizas tenga que variar el número de cifras significativas para poder medir tiempos con comodidad.

Este comando 'time' es un comando interno pero en Linux también hay un comando externo llamado de la misma forma. Para poder ejecutarlo con esta shell debería incluir el camino completo. No deseamos abusar de su escaso 'time' así que no lo comentaremos. Para buscar en el man el comando 'time' que hemos explicado o de cualquier otro comando interno tendría que mirar en la página del manual de bash.

Comando kill

Este comando se utiliza para matar procesos. En realidad envia señales a otros procesos, pero la acción por defecto asociada a la mayoria de las señales de unix es la de finalizar el proceso. La finalización de un proceso puede venir acompañada del volcado de la información del proceso en disco. Se genera un fichero 'core' en el directorio actual que solo sirve para que los programadores localicen el fallo que provocó esta prusca finalización del proceso. Por ejemplo si el proceso intenta acceder fuera del espacio de memoria concedido por el kernel, recibirá una señal que lo matará. Lo mismo ocurrirá si se produce una división por cero o algún otro tipo de error irrecuperable.

Un proceso unix puede capturar cualquier señal excepto la señal 9. Una vez capturada la señal se puede activar una rutina que puede programarse con toda libertad para realizar cualquier cosa.

'kill' por defecto es 'kill -15' envia un SIGTERM y generalmente provoca cierre ordenado de los recursos en uso. Esta señal puede ser ignorada, o puede ser utilizada como un aviso para terminar ordenadamente. Para

matar un proceso resulta recomendable enviar primero un kill -15 y si no se consigue nada repetir con kill -9. Este último -9 envia SIGKILL que no puede ser ignorada, y termina inmediatamente. Solo fallara si no tenemos permisos para matar ese proceso, pero si es un proceso nuestro, kill -9 resulta una opción segura para finalizar un proceso.

Las señales actuan frecuentemente como avisos de que ha ocurrido algo. Existen muchos tipos de señales para poder distinguir entre distintas categorías de incidencias posibles.

Comando nice
El multiproceso esta implementado concediendo ciclicamente la CPU en rodajas de tiempo a cada proceso que esta en ejecución.

Existen dos numeros de prioridad. La prioridad NICE y la prioridad concedida por el Kernel mediante un algoritmo. Esta última no tiene porque coincidir con nice y puede valer mas de 39. En cambio el comando nice solo acepta valores comprendidos entre 0 y 39, siendo 20 el valor por defecto. Cuando nice sube el valor significa que el proceso tiene baja prioridad. El comando 'nice -10' incrementara el valor nice en 10 (es decir baja la prioridad). Para bajar el valor de nice (Es decir para subir la prioridad) hace falta permisos de superusuario.

En un sistema con poca carga de trabajo no se notará apenas diferencia al ejecutar un comando con baja prioridad o con alta prioridad. Pero en un sistema con la CPU sobrecargada los comandos ejecutados con prioridad más baja se veran retrasados. ya que el kernel concederá más tiempo de CPU a los procesos con prioridad más alta.

Hay otros comandos de interés. Por ejemplo 'top' muestra los procesos que mas CPU estén consumiendo. 'vmstat' saca información del consumo de memoria virtual.

Hay que tener en cuenta que el sistema gasta recursos en la gestión de los procesos. Por ejemplo si estamos compartiendo una máquina con otros usuarios y tenemos que realizar 15 compilaciones importantes terminaremos antes haciendolas en secuencia una detras de otra que lanzandolas todas a la vez. La avaricia de querer usar toda la CPU posible para nosotros puede conducir a una situación en la cual ni nosotros ni nadie sacará gran provecho de la CPU. El sistema realizará una cantidad enorme de trabajo improductivo destinado a mantener simultanemente funcionando una gran cantidad de procesos que gastan mucha CPU y mucha memoria. La máquina comienza a usar el disco duro para suplir la falta de RAM y comienza a gastar casi todo el tiempo en el intercambio de la RAM con el disco duro. A esta situación se la denomina swaping.

Comando renice
Sirve para cambiar la prioridad de un proceso. Sigue la misma filosofia que el comando nice pero hay que identificar el o los procesos que deseamos cambiar su prioridad. Se puede cambiar la prioridad de un proceso concreto dado su PID o los procesos de un usuario dando su UID o todos los procesos pertenecientes a un determinado grupo. dando su GID. Tanto el comando nice como el comando 'renice' tienen mayor interés para un administrador de sistemas que para un usuario normal. Consulte las páginas del manual para más detalles.

SISTEMA DE FICHEROS Primera parte

- Introducción
- Nombres de ficheros
- Comandos para manejar el sistema de ficheros
- Alias para el uso de los comandos más frecuentes
- Ejemplos de uso de los comandos estudiados
- Sistema plano de ficheros
- Tipos de ficheros
- Atributos de fecha en ficheros
- Test

Introducción
Este es sin duda uno de los capítulos con mayor interés práctico.

A menudo la palabra fichero se puede usar de dos formas. Ficheros en sentido amplio como cualquier cosa que tiene nombre en un sistema de ficheros y otra como ficheros propiamente dichos que más tarde llamaremos ficheros regulares. Estos últimos son la clase de ficheros más normalitos y sirven para contener información legible o ejecutable.

Se trata de una coincidencia de nombres especialmente lamentable porque dificulta mi trabajo de explicar ahora las cosas.

Por ejemplo cuando hablamos de generalidades para los nombres de ficheros y cuando hablemos de tipos de ficheros estaremos hablando de ficheros en sentido amplio, en cambio cuando hablamos de dirigir la salida a un fichero o cuando decimos que leemos un fichero nos referimos a ficheros propiamente dichos es decir ficheros regulares que son los que se usan normalmente como contenedores de información.

Las palabras no siempre significan lo mismo usadas en diferentes contextos así que nosotros confiaremos en su capacidad de deducción y usaremos indistintamente la palabra fichero para ambas cosas aunque reconocemos que es una costumbre que no facilita las cosas, pero por desgracia verá este término utilizado en diferentes sitios usado para ambas cosas. No se preocupe y tenga paciencia al fin y al cabo si ya ha llegado hasta aquí, su sagacidad a quedado prácticamente demostrada.

Comprenderá todo esto cuando hablemos de tipos de ficheros.

Para poner a prueba una vez más su sagacidad, (santa paciencia la suya) usaremos el término ficheros en sus dos acepciones para describir una clase de ficheros llamada directorios.

Los directorios pueden considerarse como una clase de ficheros que pueden contener toda clase de 'ficheros' (en sentido general) incluidos nuevos directorios, lo cual hace que la estructura final tenga estructura similar a la de un arbol.

Las ramas de un arbol forman la estructura de todo el arbol y de forma muy similar los directorios forman el armazón de toda la estructura del sistema de ficheros.

Un sistema de ficheros en Unix tiene esta estructura de árbol y el punto de origen se llama root (raíz) y se caracteriza porque el padre de 'root' resulta ser el mismo 'root' y se simboliza por '/'.

Ya sabemos que esto es raro pero no pregunte como alguien puede ser padre de si mismo. Esto sencillamente es un truco para rematar la estructura de todo el árbol de alguna forma.

Ojo el super usuario también se le llama 'root' pero no tiene nada que ver con esto. Es solo otra molesta coincidencia.

Nombres de ficheros

Un nombre de fichero valido para el núcleo de Linux puede contener cualquier carácter salvo el caracter '/' y salvo el carter '\0' (caracter nulo). Sin embargo no es prudente usar caracteres especiales interpretados por la shell ($,",',&,#,(,),*,[,],{,}, etc..) Tampoco debe empezar por el carácter '-' que puede ser confundido con una opción. Por ejemplo si llamamos a un fichero '-r' no solo no podremos borrarlo sino que se puede confundir con la peligrosísima opción '-r' de borrado recursivo.

Cuando un usuario es dado de alta se define la shell por defecto que usará y el directorio de entrada. Este directorio de entrada al sistema será el valor de la variable $HOME.

En cada directorio hay dos entradas como mínimo que son '.' y '..' que referencian respectivamente al directorio actual y al directorio padre. (Cuando no tenemos en la variable $PATH referenciado el subdirectorio actual tendremos que ejecutar los comandos de nuestro directorio actual mediante ./comando)

El carácter '/' se utiliza para separar los diferentes componentes de un nombre de fichero y por eso no se puede usar como pare de un nombre de fichero. El carter nulo se usa como fin de cadena en en lenguaje C que es el lenguaje en el que está escrito el SO. Por eso tampoco se puede usar ese caracter como nombre de fichero.

Cuando usamos un nombre de fichero que empieza por '/' se llama nombre de camino completo o nombre de camino absoluto. Si por el contrario empieza por '.', por '..', o por un nombre se denominará nombre de camino relativo, porque su utilización depende del subdirectorio actual donde nos encontremos. La última parte del nombre después del último carácter '/' se llama nombre base. basename(1).

```
$ # Ejemplo de uso del comando basename
$ basename /usr/include/sys/signal.h

signal.h
```

Los ficheros que empiezan por '.' no tienen realmente nada de especial. Ya explicamos que la shell no expande los nombres de estos ficheros y algunos comandos como 'ls' y otros los consideran de una forma especial. Por ejemplo al usar el comando 'ls' sin parámetros no muestra esos ficheros. Por todo ello aparentan ser invisibles cuando en realidad son algunos comandos como 'ls' y el propio 'bash' quienes los tratan de forma especial. Muchos ficheros que son de uso particular de algunos programas se usan de esta forma para que el usuario pueda ignorarlos en condiciones normales.

En 'msdos' existe un concepto llamado extensión. Corresponde con un máximo de tres caracteres después del último punto en un nombre de fichero. En Unix los nombres pueden tener varios puntos y después del último punto puede haber más de tres caracteres alfanuméricos ya que no existe el concepto de extensión. De existir nombres que terminan de una cierta forma puede tener significado especial para algunos programas pero no para núcleo (kernel) y tampoco para el interprete de comandos.

Lo que si tiene un tratamiento especial a nivel de sistema operativo son los caracteres '/' que forzosamente indican los elemento de un camino formado por la sucesión de caracteres. Es decir será imposible crear un

fichero con nombre 'x/a' dentro de un directorio porque el sistema interpretaría que queremos crear un fichero 'a' dentro de un directorio 'x'.

Comandos para manejar el sistema de ficheros
Relación de comandos más importantes para manejar el sistema de ficheros.

```
ls <lista>     Muestra el contenido de directorio actual.
cd <dir>       Cambiar a un subdirectorio.
mkdir <lista-dir>      Crea uno o mas directorio.
rmdir <lista-dir>      Borra uno mas directorios.
cp <lista>     Copiar uno o mas ficheros al ultimo de la lista.
               Si mas de dos el ultimo debe ser un directorio.
mv <lista>     mover o renombrar ficheros o directorios al ultimo
               nombre de la lista.
               Si mas de dos el ultimo debe ser un directorio.
rm <lista>     borrar la lista de ficheros.
ln              Crear un enlace a un fichero
touch          Crea un fichero vacío o modifica la fecha de un fichero
pwd            muestra el camino absoluto del directorio actual.
chmod          Cambia los permisos de un fichero.
chown          Cambia el propietario de un fichero.
chgrp          Cambia el grupo de un fichero.
du <fichero>   Ocupación  de un fichero
tree           Listado recursivo
tree -d        Mostrar árbol de directorios
file <fichero> Obtiene información sobre el tipo de Ficheros
```

Recuerde que tiene a sus disposición el comando 'man' para consultar las distintas opciones de cada uno de estos comandos.

Alias para el uso de los comandos más frecuentes
El comando 'cd' es un comando interno que sirve para cambiar de directorio actual. Existe otro comando interno llamado 'alias' que sirve para algo que no tiene que ver con el sistema de ficheros y sin embargo lo mencionaremos aquí por razones prácticas.

En muchos sistemas parecerá que existe la disponibilidad de usar un comando 'll' que sirve para mostrar un directorio con abundante información. En realidad se trata de un alias. Usted puede verificar los alias definidos para su usuario tecleando 'alias' sin argumentos. Generalmente se usa para abreviar el uso de comandos muy utilizados. Esto puede resultarle de interés para personalizar su entorno.

Cuando el interprete de comandos arranca se ejecutan una serie de ficheros '/etc/profile', '~/.bash_profile', '~/.bash_login', y '~/.profile' en este orden. En '.bash_profile' suele existir una parte que pregunta por la existencia de un '~/.alias' y en caso afirmativo lo ejecuta. Este fichero '.alias' contiene una serie de comandos de alias que permanecerán disponibles durante su sesión de trabajo con esa shell.

Podría parecer que estamos obsesionados con la shell y que no sabemos hablar de otra cosa pero solo queremos proporcionarle ahora unos consejos prácticos que le faciliten el uso de su sistema. Para ello tendría que editar el fichero '.alias' presente en su $HOME y añadir las líneas siguientes:

```
alias ll='ls -lrt --color=auto'
alias ll.='ls -adlrt --color=auto .[a-zA-Z0-9]*'
alias ls='ls --color=auto'
alias ls.='ls -adrt --color=auto .[a-zA-Z0-9]*'
alias md='mkdir'
```

```
alias mv='mv -i'
alias rd='rmdir'
alias rm='rm -i'
```

Puede ejecutar el fichero '.alias' para definir todos estos alias

```
$ . .alias
```

O puede ejecutar cualquiera de estos comandos alias ahora mismo, por ejemplo

```
$ alias ll='ls -lrt --color=auto'
```

Luego teclee el comando 'alias'. Comprobará que queda definido. Teclee el nuevo comando 'll' y compruebe su efecto.

El bash solo expande los alias de la primera palabra de un comando y solo cuando no van entrecomillados. Por eso cuando definimos un alias que sustituye a un comando (Por ejemplo "ls='ls --color=auto'") tendremos que usar 'ls' entre comillas simples o dobles para si no deseamos que se produzca la expansión del alias.

Para eliminar una definición de alias utilice el comando 'unalias' En nuestro caso.

```
$ unalias ll
```

Para no tener que teclear manualmente cada vez cada uno de estos alias es para lo que sirve el fichero '.alias'.

Si no sabe editarlo tendrá que esperar a que expliquemos el manejo del editor 'vi' y volver a esta sección para modificar su fichero '.alias'. Los alias serán definidos la próxima vez que entre en una nueva sesión. Explicaremos las ventajas de usar algunos de los alias que acabamos de proponerle.

ll Los ficheros más interesantes son generalmente los más recientes y para evitar que desaparezcan durante un scroll largo los mostraremos en último lugar. La opción de color auto colorea la salida del comando únicamente si la salida ocurre en el terminal.

ll. Sirve para listar ficheros que empiezan por un punto y después tienen una letra o dígito como segundo carácter. Normalmente los ficheros de configuración de su usuario se encuentran en su $HOME y utilizan nombres de este tipo que no aparecen en la expansión de un '*'.

rm Este alias evita usar 'rm' tal cual está añadiéndole la opcíon -i para confirmar antes de borrar. Realmente útil. Es muy recomendable usar 'rm' por defecto de esta forma. Si no desea ser interrogado a cada fichero si realmente desea borrarlo o no, utilice este alias tal como está pero añadiendo la opción -f. rm -f <lista-de-ficheros> borrará silenciosamente todo lo que usted ordene que no tiene que coincidir con todo lo que usted supone que ha ordenado. En especial si no pone cuidado al usar la expansión de ordenes de la shell.

La respuesta a la pregunta de como recuperar un fichero borrado accidentalmente es... 'De ninguna forma'.

Cosas simpáticas de Unix y Linux.

Puede que sepa de alguien que en cierta ocasión recuperó una información valiosa usando trucos o herramientas especiales pero no cuente con ello. Hay programas de borrado que usan papeleras y quizás piense en poner un alias de rm para que use uno de estos programas. Esto introduce ineficiencia en el sistema pero puede hacerse a nivel de usuario. Lo más correcto es usar una buena política de copias de seguridad.

mv Se trata de mover información pero aplicamos el mismo principio que para 'rm'.

Ejemplos de uso de los comandos estudiados
Ejecutaremos paso a paso una serie de comandos y comprobaremos lo que ocurre.

```
$ # Crearemos un alias para ls con algunas opciones
$ alias ll='ls -lrt --color=auto'
$ # Creamos en /tmp una serie de directorios anidados
$ cd /tmp
$ mkdir kkkk
$ cd kkkk
$ # Consultamos en que directorio estamos situados
$ pwd

/tmp/kkkk

$ # Continuamos creando ficheros y directorios
$ echo xx > kk1
$ echo xx > kk2
$ echo xx > kk3
$ mkdir kkkk
$ # Usamos el alias por primera vez para ver el contenido
$ # del directorio actual.
$ ll

total 4
-rw-r--r--   1 acastro  acastro         3 abr 16 20:01 kk1
-rw-r--r--   1 acastro  acastro         3 abr 16 20:01 kk2
-rw-r--r--   1 acastro  acastro         3 abr 16 20:01 kk3
drwxr-xr-x   2 acastro  acastro      1024 abr 16 20:01 kkkk

$ # Continuamos creando ficheros y directorios
$ cd kkkk
$ echo xx > kk1
$ echo xx > kk2
$ echo xx > kk3
$ mkdir kkkk
$ cd kkkk
$ echo xx > kk1
$ echo xx > kk2
$ echo xx > kk3
$ cd /tmp/kkkk
$ # El alias ll corresponde con el comando ls con una serie de
$ # opciones pero podemos añadir nuevas opciones. Por ejemplo
$ # podemos hacer un listado recursivo de toda la estructura de
$ # ficheros y directorios que cuelga del directorio actual.
$ # /tmp/kkk
$ ll -R .

total 4
-rw-r--r--   1 acastro  acastro         3 abr 16 20:01 kk1
-rw-r--r--   1 acastro  acastro         3 abr 16 20:01 kk2
```

```
-rw-r--r--   1 acastro  acastro         3 abr 16 20:01 kk3
drwxr-xr-x   3 acastro  acastro      1024 abr 16 20:03 kkkk

kkkk:
total 4
-rw-r--r--   1 acastro  acastro         3 abr 16 20:03 kk1
-rw-r--r--   1 acastro  acastro         3 abr 16 20:03 kk2
-rw-r--r--   1 acastro  acastro         3 abr 16 20:03 kk3
drwxr-xr-x   2 acastro  acastro      1024 abr 16 20:04 kkkk

kkkk/kkkk:
total 3
-rw-r--r--   1 acastro  acastro         3 abr 16 20:04 kk1
-rw-r--r--   1 acastro  acastro         3 abr 16 20:04 kk2
-rw-r--r--   1 acastro  acastro         3 abr 16 20:04 kk3

$ # Con el comando tree y la opción -d podemos ver la estructura
$ # simplemente con los directorios y como están colocados.
$ tree -d

.
`-- kkkk
    `-- kkkk

    2 directories


$ # Para ver la estructura completa usamos el comando
$ # tree sin opciones. Vemos ahora tambien los ficheros.
$ tree

.
|-- kk1
|-- kk2
|-- kk3
`-- kkkk
    |-- kk1
    |-- kk2
    |-- kk3
    `-- kkkk
        |-- kk1
        |-- kk2
        `-- kk3

2 directories, 9 files

$ # Con find podemos buscar usando muchos criterios. Por ejemplo
$ # por nombre.
$ find . -name 'kk*'

./kk1
./kk2
./kk3
./kkkk
./kkkk/kk1
./kkkk/kk2
./kkkk/kk3
./kkkk/kkkk
./kkkk/kkkk/kk1
```

```
./kkkk/kkkk/kk2
./kkkk/kkkk/kk3

$ # Podemos buscar por fecha de modificación comparanda con la
$ # fecha de otro fichero. Por ejemplo que ficheros son más
$ # recientes que './kkkk/kk2' ?
$ find . -newer ./kkkk/kk2

./kkkk
./kkkk/kk3
./kkkk/kkkk
./kkkk/kkkk/kk1
./kkkk/kkkk/kk2
./kkkk/kkkk/kk3

$ # Podemos comprobar la ocupación de los directorios
$ du .

4       ./kkkk/kkkk
8       ./kkkk
12      .
```

Para borrar todo esto se suele usar la opción de borrado recursivo. No debe usar esta opción como root sin estar muy seguro de lo que hace. Esta opción borra recursivamente y es muy peligrosa porque un error puede borrar recursivamente desde un lugar equivocado un montón de información.

No es necesario borrar nada porque en /tmp la información suele borrarse durante el arranque. Si quiere intentarlo con las precauciones mencionadas teclee 'rm -fr /tmp/kkkk'. Si lo prefiere puede dejarlo sin borar ya que no ocupa casi nada y se borrará la proxima vez que se vacíe el '/tmp' que será seguramente cuando vuelva a arrancar el ordenador.

Para que los comandos anteriores puedan funcionar correctamente deberemos tener los permisos adecuados sobre los ficheros y los subdirectorios.

Evidentemente hay muchos más y no podemos explicarlos todos ahora pero quizás sea bueno mostrar una pequeña lista de algunos de ellos que veremos más adelante en posteriores capítulos.

```
sum             Obtiene un valor de checksum
cksum           Obtiene un valor de checksum largo y por tanto más seguro
gzip            Compresor de GNU
find            Buscar ficheros
tar             Empaqueta ficheros y directorios en un solo fichero.
cpio            Empaqueta ficheros y directorios en un solo fichero.
```

Sistema plano de ficheros

La estructura en forma de árbol tiene el inconveniente de que se requiere ir recorriendo el árbol para ir localizando cada uno de los elementos. Por ello se hace necesario un sistema mucho más directo.

A cada ficheros se le asocia un número para poder localizarlo directamente. Dicho número se llama inodo. Los números de inodos son el índice del sistema plano de ficheros, también llamado por ello tabla de inodos Cada fichero de cada sistema de ficheros tiene un inodo distinto salvo que se trate de un enlace rígido (hard link). Sin embargo hay inodos que no se corresponden con ningún fichero.

Un fichero en sentido estricto indica capacidad de contener una secuencia de bytes.

Un inodo indica básicamente la capacidad de tener un nombre asociado al sistema de ficheros.

Un fichero tiene asociadas funciones de posición y de lectura y escritura de información dentro del fichero. Un inodo es una referencia muy directa a un fichero y a grandes rasgos tiene asociadas funciones que permiten manejar y construir la estructura de árbol del sistema de ficheros.

Tipos de ficheros

- Regular. Son meros almacenes de información. Algunos contiene código ejecutable.
- Directorios Son una tabla con números de inodos y nombres de ficheros.
- Ficheros especiales. Pueden ser dispositivo tipo carácter o dispositivo de bloques. El manejo de estos ficheros depende del dispositivo en particular.
- Fifo son pipes con nombre. Son propios de System V y en BSD no existen pero en Linux si están disponibles.
- Enlaces simbólicos (symbolic links). Son ficheros que contiene un puntero a otro fichero que podría perfectamente estar en un sistema de ficheros distinto.
- El Soket es un fichero especial de 4.3 BSD y se utiliza para comunicar procesos que pueden estar en máquinas distintas.
- Enlaces rígidos (hard links). Realmente es un único fichero que puede ser visto con distintos nombres dentro de un mismo sistema de ficheros. Es decir se pueden observar como si fueran ficheros idénticos con el mismo inodo. La información reside en un mismo lugar y lo que ocurra aparentemente en un sitio ocurrirá instantáneamente en el otro lugar. Los enlaces de este tipo no pueden ocurrir entre sistemas de ficheros distintos. No todos los sistemas de ficheros soportan hard links ya que es un concepto muy ligado a los SO tipo Unix.

Generalmente los diferentes tipos de ficheros son capaces de aceptar operaciones de apertura, cierre, lectura y escritura de forma similar. Por ello se puede redirigir una entrada y una salida a diferentes tipos de ficheros. Parte de la enorme potencia y flexibilidad de los SO tipo Unix residen en este hecho. Gracias a esto la combinación de entradas salidas entre procesos, ficheros, dispositivos, etc.. resulta muy sencilla y flexible. Existe una salvedad importante. Los directorios en este sentido son totalmente especiales y un intendo de escribir en un directorio dará siempre un error aunque sea realizado por 'root'. No tiene sentido permitirlo. Los directorios se crean, borran, y modifican con comandos específicos para ellos.

```
$ echo > .

bash: .: Es un directorio
```

Atributos de fecha en ficheros

En Unix y Linux los ficheros en sentido amplio tienen asociadas siempre tres fechas. En realidad estas fechas están almacenadas internamentte como el número de segundos transcurridos desde el '1 de Enero de 1970'.

Una de ellas indica el tiempo del último acceso realizado sobre ese fichero. La otra indica la fecha de creación (en realidad es la fecha del último cambio de estado) de ese fichero y la última y quizás más importante (es la que vemos al consultar con 'ls -l') indica la fecha de la última modificación.

En realidad si cambiamos el estado de un fichero por ejemplo cambiando permisos o el propietario se modificarán las fechas de creación y del último acceso. Por eso la fecha que hemos denominado de creación puede ser posterior a la fecha de modificación del fichero.

Cuando se crea un fichero las tres fechas tendrán el mismo valor. Cuando se lee un fichero se modifica la fecha de acceso del mismo pero acceder al nombre de un fichero o consultar el estado de un fichero no modifica ninguna fecha. Su fecha de acceso tampoco ya que en realidad lo que se hace es leer la información del directorio que lo contiene el cual si que verá modificada su fecha de acceso.

Las fechas de modificación asociadas a los directorios cambian con las altas, o bajas de los elementos dentro del directorio. Esto es debido a que dichas operaciones se consideran como si fueran escrituras en el fichero especial de tipo directorio. Un directorio es al fin y al cabo una simple tabla. Esto lo recordaremos cuando hablemos de permisos asociados a directorios.

Para modificar las fechas de modificación y de acceso de un fichero se puede usar el comando touch(1)

SISTEMA DE FICHEROS Segunda parte

- Permisos de ficheros
- Notación numérica para permisos
- Umask
- Comando chmod
- Comando chown
- Concesión de acceso por parte del kernel
- Significado de los permisos en directorios
- Setuid y Setgid
- Test

Permisos de ficheros
Usaremos el término fichero en su sentido más amplio. Es decir que el tema de permisos es aplicable a distintos tipos de ficheros con algunas matizaciones que explicaremos más adelante. Los ficheros tienen muchos atributos además de su nombre. Para ver los más significativos haremos:

```
$ ls -l /
```

Supongamos que tenemos el siguiente fichero llamado 'kkkkk'

```
-rwxrwxrwx  1 root root  14740 abr 15 12:05 kkkkk
^^^^^^^^^^  ^  ^    ^      ^    ^   ^  ^  ^  ^
||||||||||  |  |    |      |    |   |  |  |  |__nombre del fich.
||||||||||  |  |    |      |    |   |  |  |
||||||||||  |  |    |      |    |   |  |  |__minutos : Fecha y
||||||||||  |  |    |      |    |   |  |_____hora    : hora de la
||||||||||  |  |    |      |    |   |____día del més : última
||||||||||  |  |    |      |    |________más         : modificación
||||||||||  |  |    |      |
||||||||||  |  |    |      |___________ Tamaño en bytes
||||||||||  |  |    |
||||||||||  |  |    |_______nombre del grupo
||||||||||  |  |____________nombre del propietario del fichero
||||||||||  |
```

```
||||||||||  |_______________número de enlaces rígidos (hard links)
||||||||||
||||||||||___001____________ permiso de ejecución para : Un usuario
|||||||||____002____________ permiso de escritura para : cualquiera
||||||||_____004____________ permiso de lectura para   :
|||||||
|||||||______010____________ permiso de ejecución para : Un usuario
||||||_______020____________ permiso de escritura para : pertene-
|||||________040____________ permiso de lectura para   : ciente al
||||                                                    : grupo
||||
||||________100____________ permiso de ejecución para : El usuario
|||_________200____________ permiso de escritura para : propieta-
||__________400____________ permiso de lectura para   : rio
|
|____________________________tipo de fichero
                               -  Fichero regular (fichero normal)
                               d  Directorio
                               l  Enlace simbólico
                               p  Fifo con nombre
                               b  Dispositivo de bloques
                               c  Dispositivo de caracteres
```

En inglés se usan los términos owner , group, y others para designar respectivamente al propietario, al grupo y a cualquier usuario.

Notación numérica para permisos

Recuerde que también tenemos que pensar en la posibilidad de que mi amigo David acostumbrado al uso del azadón decida hacer este curso y teníamos un compromiso de explicar las cosas partiendo de cero. Los permisos de los ficheros son almacenados en formato binario y se puede referenciar numéricamente. Vimos que a cada permiso individual le asociabamos un número de tres dígitos formado por dos ceros y un tercer número que podía ser unicamente 1, 2, o 4. Por ejemplo el permiso de escritura para un usuario cualquiera era 002. Con estos números se puede codificar los permisos de la forma que indicamos en el siguiente ejemplo:

```
r w x - - w x - r - x     Esto equivaldría a un permiso     735
4 2 1 - 0 2 1 - 4 0 1     ( 4+2+1 ,  0+2+1 , 4+0+1  = 7,3,5 )
```

Los permisos también pueden representarse como una secuencia de bits. Un bit es un valor que solo puede valer 0 o 1. En el caso anterior podríamos representarlo de la forma 111 010 101 donde 1 indica que si hay permiso 0 que no lo hay y la posición de cada 0 y cada 1 representa a cada uno de los permisos.

Umask

Es un comando interno del bash. Se utiliza cuando se crean ficheros.

No podemos profundizar mucho en temas de matemática binaria porque nos saldriamos del tema pero la umask se aplica mediante una operación llamada AND NOT.

Consiste en tomar a umask como una máscara donde los bits a 1 especifican los bits de permisos que se pondrán a cero.

Por ejemplo si queremos abrir un fichero con permisos 664 y tenemos una umask de 022 obtendremos un fichero 644.

```
664   110 110 100
022   000 010 010
-----------------
644   110 100 100
```

Esto tiene la utilidad de proteger el sistema frente a cierto tipo de descuidos durante la creacion de ficheros.

Por ejemplo supongamos que un administrador de un sistema considera que por defecto todos los ficheros que el crea deberían carecer de permisos de ejecucion y de escritura para todo el mundo y que para los usuarios de su mismo grupo de berían de carecer de permiso de ejecución. Para ello establecerá una 'umask = 023'. Es decir una umask que elimina los permisos - - - . - w - . - w x

Para un script que genere ficheros que solo deban ser accesibles para el usuario que los ha generado usaríamos. 'umask = 077'

Comando chmod

Este comando sirve para alterar una serie de atributos del fichero Existe dos formas de usarlo. Una indicando el tipo de acceso y a quien lo queremos conceder o eliminar. Existen una serie de atributos que se pueden modificar usando este comando.

```
04000   Set uid on execute.
02000   Set gid on execute.
01000   Save text on image after execution.
00400 r Permiso de lectura para el propietario   (owner)
00200 w Permiso de escritura para el propietario
00100 x Permiso de ejecución para el propietario
00040 r Permiso de lectura para el grupo         (group)
00020 w Permiso de escritura para el grupo
00010 x Permiso de ejecucion para el grupo
00004 r Permiso de lectura para cualquiera       (others)
00002 w Permiso de escritura para cualquiera
00001 x Permiso de ejecucion para cualquiera
```

Los atributos Set uid, Set gid, y Save text no los explicaremos de momento.

Consulte las páginas del manual relativas a este comando. Como de costumbre el manual resulta imprescindible a la hora de recordar la utilización de un comando pero no a la hora de comprender conceptos nuevos. No se preocupe con los ejercicios que le proponemos a continuación comprenderá perfectamente:

```
$ cd /tmp
$ echo > kk
$ ##################################
$ chmod 777 kk
$ ls -l kk

-rwxrwxrwx  .... .... ......

$ ##################################
$ chmod 707 kk
$ ls -l kk

-rwx---rwx  .... .... ......
```

```
$ ################################
$ chmod 421 kk
$ ls -l kk

-r---w---x  .... .... ......

$ ################################
$ chmod 124 kk
$ ls -l kk

---x-w-r--  .... .... ......

$ ################################
$ # 'chmod 0 kk'  equivale a 'chmod 000 kk'
$ chmod 0 kk
$ ls -l kk

----------  .... .... ......

$ ################################
$ chmod +r kk
$ ls -l kk

-r--r--r--  .... .... ......

$ ################################
$ chmod +x kk
$ ls -l kk

-r-xr-xr-x  .... .... ......

$ ################################
$ chmod -r kk
$ ls -l kk

---x--x--x  .... .... ......

$ ################################
$ chmod u+r kk
$ ls -l kk

-r-x--x--x  .... .... ......

$ ################################
$ chmod a-x kk
$ ls -l kk

-r--------  .... .... ......

$ ################################
$ chmod g+x kk
$ ls -l kk

-r----x---  .... .... ......

$ ################################
$ chmod o+x kk
$ ls -l kk
```

```
-r----x--x  .... .... ......

$ ##################################
$ chmod a+rwx kk
$ ls -l kk

-rwxrwxrwx  .... .... ......

rm kk
```

Comando chown

El comando 'chown' sirve para cambiar el UID y el GID de un fichero. Esto solo se puede hacer si tenemos los permisos para ello. Normalmente su uso está reservado a 'root' y por ello no diremos nada más. Es muy posible que si usted usa Linux en un ordenador personal necesite algunos de estos conocimientos pero se sale de los propósitos de este curso. Le basta con saber que existe y para que sirve.

Concesión de acceso por parte del kernel
Explicaremos el funcionamiento de los permisos de la forma más precisa que nos sea posible.

Para ello usaremos unas abreviaturas que ya conocemos.

- EUID es el Identificador de usuario efectivo de un proceso
- EGID es el Identificador de grupo efectivo de un proceso
- UID en un fichero es un atributo que identifica al propietario.
- GID en un fichero es un atributo que identifica al grupo del propietario.

En realidad hablamos de propietario como el usuario que creo el fichero. El Kernel realizará el siguiente test para conceder a un proceso el acceso de cierto tipo a un fichero.

1. Si el EUID del proceso es 0 se da acceso. (root puede hacer lo que sea)
2. Si el EUID del proceso es igual al UID del owner (propietario del fichero) se concede el acceso si los permisos de usuario rwx son los adecuados.
3. Si el EUID del proceso es distinto al UID del owner, y si el EGID del proceso es igual al GID del owner, se concede el acceso si los permisos de grupo rwx son los adecuados.
4. Si el EUID del proceso es distinto al UID del owner, y si el EGID del proceso es distinto al GID del owner, se concede el acceso si los permisos de others rwx son los adecuados.

NOTA el comando rm permite borrar cualquier fichero sea cual sean los permisos cuando el proceso tiene un EUID coincidente con el propietario del fichero. Unicamente ocurre que si el fichero esta protegido contra escritura suele pedir (dependiendo de la configuración) confirmacion antes de borrarlo. El permiso para borrar un fichero no se guarda como atributo del fichero. Esto se hace a nivel de directorio y pasamos a explicarlo inmediatamente.

Significado de los permisos en directorios
Para entender como funcionan los permisos aplicados a un directorio hay que imaginar que un directorio es un fichero normal que solo contiene una tabla en la que se relacionan los ficheros presentes en ese directorio. En todos los directorios existen siempre un par de entradas obligadas que son '.' y '..' para el directorio actual y para el directorio padre respectivamente. Vamos a explicarlo sobre la marcha a la vez que hacemos el ejercicio. Recuerde usar un usuario normal (distinto de root) para realizar los ejercicios.

```
$ cd /tmp
$ mkdir kk
$ cd kk
$ echo > k1
$ echo > k2
$ cd /tmp
$ chmod 777 kk
$ ## Para ver como ha quedado los permisos hacemos ...
$ ls -ld kk

drwxrwxrwx   2 ..... ...... kk

$ ## Para ver que ficheros contiene el directorio kk hacemos ...
$ ls  kk

k1 k2
```

Si un directorio no tiene permiso de lectura, resultara imposible para cualquier comando incluido ls averiguar cual es su contenido.

Sin abandonar la sesión anterior continuamos el ejercicio

```
$ ###(1)###
$ ## Eliminamos los premisos de lectura del directorio kk
$ chmod -r kk
$ ls -ld kk

d-wx-wx-wx   2 ..... ...... kk

$ ls -l kk

ls: kk: Permiso denegado
$ ## En este intante podemos entrar dentro del directorio
$ ## Incluso podemos crear un nuevo fichero pero ya no podemos
$ ## Saber cual es el contenido del directorio.
$ cd kk
$ pwd

/tmp/kk

$ echo > k3
$ ls -l

ls: .: Permiso denegado

$ ## Hemos podido meternos dentro del directorio kk porque aun
$ ## Tenemos permiso para ello pero seguimos sin poder saber
$ ## cual es su contenido.
$ cd /tmp
```

Si no se tiene permiso de ejecucion en un directorio no se podra hacer cd a ese directorio, ni a ninguno de los directorios que cuelgan de el. Esto imposibilita todas las operaciones que utilicen ese directorio como parte del camino

Sin abandonar la sesión anterior continuamos el ejercicio

```
$ ###(2)###
$ ## Ahora eliminamos permiso de ejecución
$ chmod -x kk
$ ls -ld kk

d-w--w--w-   2 ..... ...... kk

$ ## Intentamos entrar
$ cd kk

bash: cd: kk: Permiso denegado

$ ## No hemos podido entrar
$ rm kk/k2

bash: cd: kk: Permiso denegado

$ ## Tampoco nos deja borrar el fichero desde fuera
$ #
$ ## Recuperamos permiso de ejecución
$ chmod +x kk
$ ls -ld kk

d-wx-wx-wx   2 ..... ...... kk

$ rm kk/k2
$ ls kk

k1
```

Si no tiene permiso de escritura no se podra dar altas, baja o modificaciones en la tabla lo que se traduce en que no se pueden borrar sus ficheros ni crear otros nuevos, ni renombrar ficheros.

Sin abandonar la sesión anterior continuamos el ejercicio

```
$ ###(3)###
$ ## Ahora eliminaremos permisos de escritura
$ chmod 666 kk
$ ls -ld kk

dr-xr-xr-x   2 ..... ...... kk

$ ## Ahora no tenemos permiso de escritura
$ cd kk
$ ls

rm: ¿borrar `k1'? (s/n)  s
rm: k1: Permiso denegado

$ echo > k3

bash: k3: Permiso denegado
```

```
$ touch k3

touch: k3: Permiso denegado

$ ## Sin permiso de escritura en el directorio no se pueden
$ ## Crear ficheros nuevos dentro de el ni borrar ficheros
$ ## Pero los ficheros son perfectamente accesibles
$ echo "Mensaje de prueba" > k1
$ cat k1

Mensaje de prueba

$ ## Vamos a limpiar
$ chmod 777 /tmp/kk
$ rm /tmp/kk/*
$ rmdir /tmp/kk
```

Setuid y Setgid

Cada proceso tiene un (R)UID, (R)GID, EUID, y EGUID. Cada fichero tiene un UID y un GID owner.

El comando 'chmod 4000' pone el bit setuid . Esto hace que el proceso que ejecuta este comando temporalmente cambie su EUID tomando el valor UID del fichero. Es decir el proceso sufre un cambio de personalidad y se convierte en la persona que figura como propietaria de ese fichero. Para ello lógicamente ese fichero deberá ser un ejecutable. El comando passwd sirve para cambiar la clave. Se trata de un comando propiedad de root con setuid. Por lo tanto mientras estamos ejecutando este comando adoptamos la personalidad de 'root'. Gracias a ello se nos permite acceder a la clave para modificarla y gracias a que ese comando no permite hacer nada más que eso se evita que este proceso con personalidad de 'root' pueda hacer otras cosas.

Busque el comando 'passwd' en su sistema. Podría estar en '/usr/bin/passwd' pero si no se encuentra en ese sitio puede localizarlo con 'which passwd'

Haga un ll para comprobar sus permisos y verá que es propiedad de 'root' y que tiene una 's' en lugar de una 'x' en la parte de los permisos para el propietario.

```
-rwsr-xr-x   1 root     root        28896 jul 17  1998 /usr/bin/passwd
```

El comando 'chmod 2000' pone el bit setgid on execute. Hace algo parecido al anterior pero a nivel de grupo. Es decir el proceso que ejecuta este comando temporalmente cambie su EGID tomando el valor GID del fichero. Por ejemplo el comado lpr no solo nos convierte en 'root' cuando lo ejecutamos sino que nos cambia el grupo por el grupo de impresión.

```
-rwsr-sr-x   1 root     lp          14844 feb  7  1999 /usr/bin/lpr
```

El comando 'chmod 1000' pone el bit sticky. Se usa poco y tenía más utilidad en sistemas Unix más antigüos. Sirve para que el proceso permanezca en memoria RAM todo el rato mientras se esté ejecutando. Normalmente cuando muchos procesos usan la memoria RAM se van turnando en su uso y se descargan a disco para ceder su espacio en memoria RAM a otros procesos. Con el uso de este bit se evita la descarga a disco. Es una forma de aumentar la velocidad de ese proceso.

Para ver el aspecto de estos permisos realice el siguiente ejercicio. Realice siempre los ejercicios desde un usuario normal distinto de root.

```
$ cd /tmp
$ echo  > kk
$ chmod 4000 kk
$ ll kk

---S------   1 ..... ..... kk

$ chmod 4100 kk

---s------   1 ..... ..... kk

$ chmod 2000 kk

------S---   1 ..... ..... kk

$ chmod 2010 kk

------s---   1 ..... ..... kk

$ chmod 1000 kk

---------T   1 ..... ..... kk

$ chmod 1001 kk

---------t   1 ..... ..... kk
```

SISTEMA DE FICHEROS Tercera parte

- Tipos de enlaces (links) y su manejo
- Montage de dispositivos
- Directorio /proc
- Sistema de ficheros virtual
- Estructura estandar del sistema de ficheros de Linux
- Test

Tipos de enlaces (links) y su manejo

La palabra link es inglesa y para el concepto que vamos a tratar podríamos traducirla por enlace. Hay dos tipos de enlaces llamados 'hard link' y 'symbolic link'. Podriamos traducirlo por enlace rígido y por enlace simbólico. El término enlace simbólico se usa con frecuencia en español y parece muy adecuado pero realmente no existe una traducción para hard link tan aceptada. Nosotros emplearemos el término rígido para hard que literalmente significa duro.

Ya hemos mencionado algunas cosas sobre ellos pero sin profundizar demasiado.

El manejo de ambos tipos de enlaces se hace con el comando 'ln'. Como es habitual un repaso a la página man de este comando es muy recomendable. La creación de un enlace rígido y uno simbólico es muy similar. La diferencia es que para el enlace simbólico tendremos que usar la opción -s.

Como es habitual asumimos que las prácticas se realizarán desde un usuario distinto de root.

```
$ cd /tmp
$ mkdir /tmp2
$ cd tmp2
$ echo xxxx > ej0
$ ## Ahora creamos un par de enlaces rígidos con ej0
$ ln ej0 ej1
$ ln ej0 ej2
$ ## Creamos también un enlace simbólico ejs1 a ej0
$ ln -s ej0 ejs1
$ mkdir dir1
$ ## También creamos un enlace simbólico dir2 a dir1
$ ln -s dir1 dir2
$ ls -l

drwxr-xr-x   2 pepe user 1024 may 18 17:28 dir1
lrwxrwxrwx   1 pepe user    4 may 18 17:28 dir2 -> dir1
-rw-r--r--   3 pepe user    1 may 18 17:26 ej0
-rw-r--r--   3 pepe user    1 may 18 17:26 ej1
-rw-r--r--   3 pepe user    1 may 18 17:26 ej2
lrwxrwxrwx   1 pepe user    3 may 18 17:28 ejs1 -> ej0
```

Con esto acabamos de crear dentro de tmp un directorio tmp2 y dentro de el hemos creado ya algunos ficheros, algunos directorios y unos cuantos enlaces rígidos y simbólicos. Los enlaces rígidos no muestran nada particular listados con 'ls -l' pero los enlaces simbólicos vienen acompañados de una flecha que apunta a otro nombre de fichero.

El fichero 'ej0' lo hemos creado con un contenido 'xxxx' para ver que pasa con ese contenido más adelante. El nombre de usuario 'pepe' y el nombre de grupo 'users' son ficticios y en su sistema obtendrá otra cosa. Hay una columna de números a continuación de los permisos. Se trata de una columna que indica el número de enlaces rígidos que están asociados a un mismo fichero. En el caso de 'ej0', 'ej1', 'ej2' aparece un 3 y está claro porque son enlaces creados por nosotros pero el directorio 'dir1' tiene un 2. Esto significa que existe otro enlace rígido para ese directorio que nostros no hemos creado. Se ha creado automáticamente al crear 'dir1'. Acuerdese que todos los directorios se crean con un par de entradas que son '.' y '..' El 2 por lo tanto en este caso se debe a la entrada '.' dentro del propio 'dir1' y si dentro de 'dir1' existieran directorios habría que contabilizar cada uno del los '..' de los directorios hijos como enlaces rígidos de 'dir1'. Un fichero se corresponde con un único inodo. Es decir tiene una única entrada en la tabla plana del sistema de ficheros, pero quizás aparezca varias veces en distintas partes del sistema de ficheros, o con distinto nombre. Si esto último no le ha quedado claro vuelva a leerlo después de finalizar este capítulo porque vamos a seguir exiplicando que es un enlace rígido. Ahora retomamos la práctica en el punto donde la dejamos y continuamos.

```
$ cat ej0

xxxx

$ echo kkkkkkkkk > ej1
$ cat ej0

kkkkkkkkk

$ cat ejs1

kkkkkkkkk
```

Vemos que el contenido de los distintos enlaces con 'ej0' es idéntico, y si modificamos el contenido de cualquiera de ellos se afectará instantaneamente el contenido de los restantes. En realidad la información es accesible desde distintos nombres de ficheros pero no son copias sino que se trata de la misma unidad de información. Continuamos con el ejercicio.

```
$ rm ej0
$ cat ejs1

cat: ejs1: No existe el fichero o el directorio

$ cat ej1

kkkkkkkkk
```

Aquí ya vemos una diferencia. Pese a que 'ej1' se creó como un enlace de 'ej0', 'ej1' mantiene accesible la información incluso aunque desaparezca 'ej0'. Eso es porque en el caso de los enlaces rígidos da igual cual es el enlace o fichero original. Son totalmente equivalentes y la información solo desaparecerá del sistema cuando el último enlace rígido sea eliminado. La diferencia con el enlace simbólico es que actua simplemente accediendo al nombre del fichero que tiene almacenado en su interior. Por eso en el caso que acabamos de ver 'ejs1' queda apuntando a 'ej0' que es un fichero que ya no existe. Continuamos con el ejercicio y ahora usaremos una opción para 'ls' que no habíamos visto antes. Se trata de la opción -i que sirve para visualizar el número de inodo.

Un inodo es una clave numérica para el acceso al sistema plano de ficheros donde cada punto capaz de recibir o entregar información, tiene una única clave. Por eso los distintos enlaces rígidos contienen el mismo valor de inodo. Lo de recibir o entregar información se traduce en chorros de bytes producidos en operaciones de lectura escritura pero estas funciones pese a manejarse siempre igual realizan cosas muy distintas dependiendo del tipo de fichero. Un fichero regular es un almacén de información. Otros tipos de ficheros no son meros contenedores de bytes. Por ejemplo los dispositivos especiales de bloques o de caracteres pueden verse como emisores y receptores de bloques o caracteres respectivamente asociados a un dispositivo. Tampoco son meros contenedores de bytes los sockets, enlaces simbólicos, fifos con nombre, etc. Salvo los directorios, todos son capaces de recoger chorros de bytes o de entregar chorros de bytes o ambas cosas a la vez y lo más importante, todos ellos tienen asociado un inodo y al menos un nombre de fichero colgando de alguna parte de sistema de ficheros. Los directorios no se pueden manejar con funciones clásicas de lectura escritura. Tienen sus propias funciones específicas para manejo de directorios.

```
$ ls -li

  73449 drwxr-xr-x   2 pepe user 1024 may 18 17:28 dir1
  59173 lrwxrwxrwx   1 pepe user    4 may 18 17:28 dir2 -> dir1
  59171 -rw-r--r--   2 pepe user   10 may 18 17:30 ej1
  59171 -rw-r--r--   2 pepe user   10 may 18 17:30 ej2
  59172 lrwxrwxrwx   1 pepe user    3 may 18 17:28 ejs1 -> ej0
```

Como se puede ver 'ej1' y 'ej2' tienen el mismo valor de 59171 que en su sistema será otro valor cualquiera. En este momento después de borrar 'ej0' figuran con el valor 2 para el número de enlaces rígidos asociados. Vamos a mover el enlace simbólico 'ejs1' a 'dir1' pero vamos a usar 'mv ejs1 dir2' en lugar 'mv ejs1 dir1' pero debería dar lo mismo ya que 'dir2' es un enlace simbólico a 'dir1'.

```
$ mv ejs1 dir2
$ #
$ ## Comprobamos el resultado
$ ls -li

  73449 drwxr-xr-x   2 pepe user 1024 may 18 17:32 dir1
  59173 lrwxrwxrwx   1 pepe user    4 may 18 17:28 dir2 -> dir1
  59171 -rw-r--r--   2 pepe user   10 may 18 17:30 ej1
  59171 -rw-r--r--   2 pepe user   10 may 18 17:30 ej2

$ ls dir2

ejs1

$ ls -li dir1

  59172 lrwxrwxrwx   1 pepe user    3 may 18 17:28 ejs1 -> ej0
```

Hemos comprobado que un enlace simbólico se ha comportado igual que si fuera el propio directorio apuntado. En realidad podemos actuar a todos los efectos como si se tratara del verdadero fichero en lugar de un enlace simbólico salvo en el momento de su creación y en el momento de su destrucción. La operación 'rm' sobre un fichero simbólico no actua sobre el fichero apuntado sino sobre el propio enlace simbólico destruyendolo.

Ahora tenemos 'dir2/ejs1 -> ej0' pero 'ej0' ni siquiera existe. Vamos a cambiar el nombre de 'ej2' que era un enlace rígido de 'ej0' por 'ej0'.

```
$ mv ej2 ej0
$ cat dir2/ejs1

cat: dir2/ejs1: No existe el fichero o el directorio
```

Bueno este error es lógico porque el enlace 'dir2/ejs1' apunta a 'ej0' y no importa que estemos en el mismo directorio que 'ej0' sino que 'dir2/ejs1' y 'ej0' estén en el mismo directorio. Los enlaces son siempre relativos al sitio donde se encuentra el propio enlace. En caso contrario podría resultar graciosísimo pero poco práctico porque dependiendo de donde estuvieramos nosotros (más exactamente dependiendo del directorio actual del proceso que lo use) apuntaría a un directorio distinto cada vez.

De todas formas comprobemoslo trasladando 'ej0' a 'dir2', y observando como ha quedado todo.

```
$ mv ej0 dir2
$ cat dir2/ejs1

kkkkkkkkk

$ ls -li

  73449 drwxr-xr-x   2 pepe user 1024 may 18 17:34 dir1
  59173 lrwxrwxrwx   1 pepe user    4 may 18 17:28 dir2 -> dir1
  59171 -rw-r--r--   2 pepe user   10 may 18 17:30 ej1

$ ls -li dir2
```

```
  59173 lrwxrwxrwx   1 pepe user    4 may 18 17:28 dir2 -> dir1

$ rmdir dir2

rmdir: dir2: No es un directorio

$ rm dir2
$ ls -li

  73449 drwxr-xr-x   2 pepe user 1024 may 18 17:34 dir1
  59171 -rw-r--r--   2 pepe user   10 may 18 17:30 ej1

$ ls -li dir1

  59171 -rw-r--r--   2 pepe user   10 may 18 17:30 ej0
  59172 lrwxrwxrwx   1 pepe user    3 may 18 17:28 ejs1 -> ej0
```

Montage de dispositivos

Pueden existir varios sistemas de ficheros cada uno en un dispositivo o partición distinta, pero para hacerlo accesible ha de ser montado dentro del sistema principal de ficheros. Para ello se utilizan los comandos 'mount' y 'umount'.

Explicaremos esto un poco más. Supongamos que dispone de un disquete en el que desea guardar información accesible como parte del sistema actual. En MSDOS se accedería a la unidad que representa este dispositivo mediande el comando a: o b: por ejemplo. En Unix esto no ocurre porque ese patético invento llamado unidad lógica no existe.

Perdón por lo de patético pero es que en MSDOS, y en Windows cambias las cosas de sitio y terminas reinstalando todo.

Supongamos que tenemos en un dispositivo cdrom, cinta, disquete, o lo que sea la siguiente estructura de directorios.

```
.
|-- Guia-del-enROOTador-2.8
|-- curso
|-- glup_0.6-1.1-html-1.1
|-- lipp-1.1-html-1.1
|-- man_instal_debian_21
|-- novato-a-novato
`-- rhl-ig-6.0es
    |-- cpps
    `-- icons
```

Para poder acceder a esta información tendríamos que realizar una operación de montado sobre algún punto de nuestro sistema de ficheros actual. Por ejemplo si disponemos de un directorio '/misc/novato' podriamos montar en ese punto nuestro dispositivo y en ese caso la operación sería como enganchar un arbol en la rama de otro arbol. De esta forma para el sistema sería como si el arbol creciera.

```
.
`-- misc
    `-- novato
```

```
|-- Guia-del-enROOTador-2.8
|-- curso
|-- glup_0.6-1.1-html-1.1
|-- lipp-1.1-html-1.1
|-- man_instal_debian_21
|-- novato-a-novato
`-- rhl-ig-6.0es
    |-- cpps
    `-- icons
```

Si el administrador considera que montar y un dispositivo no entraña riesgos para el sistema concedera permisos para que esto pueda ser realizado por los usuarios. Por ejemplo un administrador puede considerar que los usuarios solo puedan montar la unidad de cdrom.

En un entorno multiusuario y multiproceso no interesa hacer efectiva las actualizaciones sobre un dispositivo de forma instantanea sino que se intenta optimizar los accesos al dispositivo. Por eso si escribimos en un dispositivo de lectura escritura como por ejemplo un disquete y sacamos dicho disquete antes de desmontarlo lo más probable es que algunas operaciones de escrituras queden sin realizar porque estaban simplemente guardadas en memoria a la espera de ser realizadas. Cuando se desmonta un sistema de ficheros se escriben las operaciones pendientes sobre ese dispositivo y entonces puede ser extraido del sistema sin peligro.

Directorio /proc

Este directorio es un directorio muy especial. Es un directorio en el cual está montado un sistema de ficheros virtual. En realidad muestra información que no reside en ningún dispositivo sino que es elaborada por el propio kernel. Por ello se trata de falsos archivos. Su ocupación en disco es 0. Sirve para comprobar la configuración y el funcionamiento del kernel. No entraremos ahora a comentarlo en detalle porque este tema es de interés para administradores del sistema. Lo que nos interesa comentar es que aquí la presentación en forma de sistema de ficheros es una simulación y se hace así para que pueda ser manejada exactamente de la misma forma que si realmente estuviera contenida en directorioss y ficheros.

Algunos de estos ficheros son de solo lectura otros son de lectura y escritura y permiten cambiar la configuración del kernel sin detenerlo. Evidentemente todo ello está configurado con los permisos adecuados para mantener la seguridad del sistema.

Si miramos dentro del directorio '/proc' veremos que hay un montón de directorios cuyo nombre es un número. Estos directorios representan cada uno a un proceso por su pid. Ya hemos estudiado el proceso init en capítulos anteriores así que vamos a poner un ejemplo.

Pruebe a hacer lo siguiente:

```
$ cat /proc/1/status
```

Obtendrá la información de estado más importante del proceso init. Además de contribuir a la cultura general sobre nuestro sistema lo que hemos mencionado sobre '/proc' nos sirve para introducir el siguiente tema.

Sistema de ficheros virtual

En Linux un sistema de ficheros puede corresponderse físicamente y lógicamente con cosas muy distintas. Acabamos de ver que el directorio '/proc' aparentemente está organizado como si fuera un sistema de ficheros idéntico a los que residen en disco duro y sin embargo se trata de algo totalmente distinto. Un sistema de ficheros puede residir, en memoria, en una partición de disco duro, en un dispositivo raid formado por varios

discos duros funcionando en paralelo y con un sistema de redundancia, en disquetes, en cdroms, en un dispositivo remoto conectado a red, etc.. También se puede implementar un sistema de ficheros dentro de un fichero. Por ejemplo umsdos es un sistema de ficheros linux implementado dentro de un fichero msdos. Realmente es una solución muy poco eficiente pero pone de relieve la flexibilidad del sistema virtual de ficheros. Un sistema de ficheros puede tener un formato interno que no siempre es el mismo. Linux puede manejar sistemas de ficheros de otros sistemas operativos. En resumen lo que Linux ofrece con su sistema ficheros virtual (VFS) es un sistema unificado de acceso a toda una variedad de recursos muy distintos. Esto es posible porque se definen una serie de funciones para manejar un sistema de ficheros genérico. En Unix cada sistema de ficheros tiene asociado un sistema plano de ficheros. Las estructuras pueden ser diferentes para cada tipo de sistema de ficheros, pero las funciones que lo manejan quedan unificadas por el (VFS)

Estructura estandar del sistema de ficheros de Linux

Existe un documento 'fsstnd' donde se describe una recomendación para estandarizar la estructura del sistema de ficheros de Linux para las distintas distribuciones. Nosotros vamos a resumir brevemenente la informacióm más importante. Un detalle mayor sería necesario si este curso fuera un curso de administración del sistema.

Este último capítulo no explica conceptos nuevos pero servirá para que comprenda lo que tiene en su sistema. Puede investigar todo lo que quiera por su cuenta siempre que use un usuario normalito sin privilegios. Si alguna vez se siente perdido solo tiene que introducir el comando 'cd' para volver a casita.

Puede haber diferencias importantes en la estructura general del sistema de ficheros entre unas distribuciones y otras pero en lineas generales el aspecto de la disposición de los directorios más importantes sería más o menos el siguiente;

```
.
|-- bin
|-- sbin
|-- tmp
|-- boot
|-- dev
|-- etc
|-- home
|-- lib
|   |-- modules
|   `-- security
|-- home
|   |-- ftp
|   |-- httpd
|   |-- luis
|   |-- msql
|   |-- pili
|   `-- skeleton
|-- root
|-- usr
|   |-- bin
|   |-- sbin
|   |-- X11R6
|   |   |-- bin
|   |   |-- include
|   |   |-- lib
|   |   `-- man
|   |-- local
|   |   |-- bin
|   |   |-- doc
|   |   |-- man
```

```
|   |   |-- lib
|   |   |-- src
|   |   `-- tmp
|   |-- doc
|   |-- include
|   |-- info
|   `-- src
|       `-- linux
|-- proc
`-- var
    |-- cache
    |-- catman
    |-- lib
    |-- local
    |-- lock
    |-- log
    |-- run
    |-- spool
    |   |-- cron
    |   |-- lpd
    |   |-- mail
    |   `-- mqueue
    `-- tmp
```

Si observa diferencias con la estructura de ficheros de su distribución no debe preocuparse esto es solo un ejemplo. Comentaremos el cometido de los directorios más significativos.

El directorio raiz

Para arrancar el sistema, debe estar presente lo suficiente como para montar '/usr' y otras partes no-esenciales del sistema de archivos. Esto incluye herramientas, información de configuración y del cargador de arranque (boot loader) y alguna otra información esencial al arrancar.

Para habilitar la recuperación y/o la reparación del sistema, estará presente en el sistema de archivos raíz aquellas herramientas que un administrador experimentado necesitaría para diagnosticar y reconstruir un sistema dañado.

Los errores del disco, que corrompen la información en el sistema de archivos '/' son un problema mayor que los errores en cualquier otra partición. Un sistema de archivos '/' (raiz) pequeño es menos propenso a corromperse como resultado de un fallo del sistema.

La principal preocupación que se usa para balancear las anteriores consideraciones, que favorecen el colocar muchas cosas en el sistema de archivos raíz, es la de mantener '/' (raiz) tan pequeno como sea razonablemente posible.

/ --- El Directorio Raíz

```
bin             Binarios de comandos esenciales
boot            Archivos estáticos de cargador de arranque(boot-loader)
dev             Archivos de dispositivos
etc             Configuración del sistema local-máquina
home            Directorios home de los usuarios
lib             Librerías compartidas
mnt             Punto de montaje de particiones temporales
root            Directorio hogar del usuario root
```

```
sbin            Binarios del sistema esenciales
tmp             Archivos temporales
usr             Segunda jerarquía mayor
var             Información variable
```

La jerarquía /usr.

'/usr' es la segunda mayor sección del sistema de archivos. '/usr' es información compartible, de solo lectura, esto significa que '/usr', debe ser compartible entre varias maquinas que corren LINUX y no se debe escribir. Cualquier información que es local a una máquina o varía con el tiempo, se almacena en otro lugar.

Ningún paquete grande (como TeX o GNUEmacs) debe utilizar un subdirectorio directo bajo '/usr', en vez, debe haber un subdirectorio dentro de '/usr/lib' (o '/usr/local/lib' si fué instalado completamente local) para ese propósito, con el sistema X Window se hace una excepción debido a un considerable precedente y a la práctica ampliamente aceptada.

```
/usr --- Segundo mayor punto de montaje (permanente)

X11R6           Sistema X Window Version 11 release 6
X386            Sistema X Windows Version 11 release 5 en plataformas  X 86
bin             La mayoría de los comandos de usuario
dict            Listas de palabras
doc             Documentación miscelánea
etc             Configuración del Sistema (todo el site)
games           Juegos y binarios educacionales
include Archivos header incluidos por programas C
info            Directorio primario del sistema GNU Info
lib             Librerías
local           Jerarquía local (vacía justo después de la instalación principal)
man             Manuales en línea
sbin            Binarios de Administración del Sistema No-Vitales
share           Información independiente de la arquitectura
src             Código fuente
```

/usr/local: Jerarquía local

La jerarquía '/usr/local' está para ser utilizada por el administrador del sistema cuando se instale el software localmente. Necesita estar a salvo de ser sobreescrito cuando el software del sistema se actualiza. Puede ser usado por programas y por información que son compartibles entre un grupo de máquinas , pero no se encuentran en '/usr'.

```
/usr/local      Jerarquía local.

bin             Binarios solo-locales
doc             Documentación local
etc             Binarios de configuración solo-local
games           Juegos instalados localmente
lib             Librerías para /usr/local
info            Páginas de info local
man             Jerarquías de páginas de manual para /usr/local
sbin            Administración del sistema solo-local
scr             Código fuente local.
```

Este directorio debe estar vacío al terminar de instalar LINUX por primera vez. No debe haber excepciones a la regla , excepto quizá los subdirectorios vacíos listados. **La Jerarquía /var**

El sistema necesita con frecuencia una zona de trabajo temporal. Si solo se requiere usar un espacio por un corto periodo de tiempo se puede usar '/tmp', pero muchas veces la información conviene manejarla y almacenarla en un lugar más permanente. El sistema puede sufrir una caida repentina y el contenido de '/tmp'

puede ser borrado durante el arranque o depurado regularmente mediante algúna tarea periódica. Por el contrario '/var' contiene todo tipo de información alguna de ella puede ser importante. Por ejemplo información vital para el mantenimiento de la gestión de paquetes de la distribución o mensajes pendientes de ser enviados por correo, archivos y directorios en fila de ejecución, información de bitácora administrativa y archivos temporales y transitorios aunque se asume que su permanencia será mayor que '/tmp'

```
/var   Información variable

       catman          Páginas del manual formateadas localmente
       lib             Información del estado de aplicaciones
       local           Información variable del software de /usr/local
       lock            Archivos de bloqueo
       log             Archivos de bitácora
       named           Archivos DNS, solo red
       nis             Archivos base de datos NIS
       preserve        Archivos almacenados después de una falla de ex o vi
       run             Archivos relevantes a procesos ejecutándose
       spool           Directorios de trabajos en fila para realizarse después
       tmp             Archivos temporales, utilizado para mantener /tmp pequeño
```

Recuerde que un número entre paréntesis a continuación de un comando indica el número de sección en el man.

Dado que no vamos a mencionar en este momento ningún comando interno del interprete de comandos bash lo que si podemos indicarle es la forma de saber cuales son y como obtener información sobre cualquiera de ellos. Bastará con teclear el comando 'help'. Si quiere obtener más información sobre cualquiera de ellos teclee 'help comando'.

Ejemplo:

```
$ help | less

$ help cd
```

Dijimos que no explicaríamos ningún comando interno pero en realidad 'help' es un comando interno y tenemos que añadir que admite el uso de meta caracteres.

```
$ help help

$ help *alias
```

No se va a estudiar en detalle ningún comando en este capítulo ya que se trata de tener una visión de conjunto. Por ello cuando lleguemos a un comando que merezca ser explicado con mayor profundidad nos limitaremos a decir para que sirve y le dejaremos con la miel en los labios con un oportuno "Se estudiará más adelante".

Selección de los comandos externos de usuario más útiles.

apropos at atq atrm *awk* banner batch bc cal cat chgrp chmod chown cksum clear cp *cpio* cut date df diff du echo *egrep* emacs env ex expr false *fgrep* file *find* finger free ftp fuser *gawk* *grep* gzip head hostname id info ispell kill killall less ln locate lpq lprm ls mail man mkdir more mv nice nohup passwd paste pr ps pstree pwd renice reset rm rmdir rsh script *sed* sleep *sort* split stty su tail talk tee telnet *test* tload top tr true *vi* w wc whatis whereis who whoami write xargs zcat zdiff zgrep zless zmore

Los comandos awk, cpio, egrep, fgrep, find, gawk, grep, sed, sort, test, y vi serán tratados en detalle en capítulos posteriores.

---

apropos(1)

Ya comentamos este comando en el capítulo dedicado al manual online de unix. Sirve para ayudar a localizar un comando que no sabemos exactamente como se escribe. Solo se usa para la búsqueda la descripción corta que figura al principio de la página del manual y hay que usar palabras completas y no secuencias de caracteres. Por ejemplo si nuestro manual está en ingles podríamos buscar compresores con:

```
$ apropos compress
```

---

at(1)

Se utiliza para programar un comando para que se ejecute en un determinado momento. Este comando no admite intervalos de tiempo menores a un minuto. La salida deberá estar redirigida ya que se ejecutará en segundo término sin un terminal asociado. Una vez ejecutado este comando devuelve un número de tarea y el momento en el cual se activará el comando. Ejemplos:

```
$ echo 'date > /tmp/salida' | at now + 1 minute
$ echo 'date > /tmp/salida' | at 8:15am Saturday
$ echo 'date > /tmp/salida' | at noon
$ echo 'echo feliz año nuevo > /dev/console' | at 11:59 Dec 31
$ echo 'banner a comer > /dev/console' | at 1:55pm
```

---

atq(1)

Permite ver la cola de tareas pendientes.

```
$ atq
```

---

atrm(1)

Permite cancelar una tarea programada y pendiente de ejecutarse más tarde.

```
$ atrm 188
```

---

awk(1)

Se trata de un programa que implementa un lenguaje para tratar ficheros de texto estructurados en campos y registros. Es muy potente y se estudiará más adelante.

---

banner(1)

Para generar cabeceras con letras grandes. Ejemplo:

```
$ banner hola


 #    #   ####   #         ##
 #    #  #    #  #        #  #
 ######  #    #  #       #    #
 #    #  #    #  #       ######
 #    #  #    #  #       #    #
 #    #   ####   ######  #    #
```

---

batch(1)

Es un comando muy similar al comando at pero no requiere la indicar el momento en ese caso se ejecutará en un momento con baja carga. Ejemplo:

```
$ echo 'date > /dev/console' | batch
```

---

bc(1)

Es una calculadora de precisión arbitraria. (Es decir con toda la precisión que uno quiera) Se puede usar interactivamente o admitir comandos por la entrada estándar. Admite expresiones bastante complejas y sentencias condicionales y de repetición que constituyen un potente lenguaje de programación para esta calculadora. Por ejemplo calcular el número PI con 300 cifras decimales.

```
$ echo "scale=300; 4*a(1)" | bc -l

3.14159265358979323846264338327950288419716939937510582097494459230 7\
8164062862089986280348253421170679821480865132823066470938446095505 8\
2231725359408128481117450284102701938521105559644622948954930381964 4\
```

```
28810975665933446128475648233786783165271201909145648566923460348610\
45432664821339360726024914127 2
```

---

cal(1)

Calendario Ejempos:

```
$ cal

$ cal 2000
```

---

cat(1)

Lee uno o más ficheros y los saca por la salida estándar.

```
$ cat fichero1 fichero2 > fichero1_mas_2
```

---

chgrp(1)

Permite cambiar el atributo de grupo de uno o más ficheros. Solo el propietario del fichero o un usuario privilegiado puede usarlo.

---

chmod(1)

Permite el cambio de los permisos asociados a un fichero. Solo el propietario del fichero o un usuario privilegiado puede usarlo.

```
$ chmod +r /tmp/fich
$ chmod u+r /tmp/fich
$ chmod 770 /tmp/fich
$ chmod a-wx,a+r /tmp/fich
```

---

chown(1)

Permite cambiar el propietario actual de un fichero. Solo el propietario del fichero o un usuario privilegiado puede usarlo. Tambien permite cambiar el grupo.

```
$ chown usuario fichero
$ chown usuario:grupo fichero
```

---

cksum(1)

Obtiene un código (CRC) que está diseñado para detectar variaciones en la información por problemas en la grabación o transmisión de datos.

```
$ cksum fichero
```

---

clear(1)

Limpia el contenido de la consola.

---

cp(1)

Copia ficheros. Si se utiliza con un par de argumentos tomará el inicial como origen y el final como destino. Si el segundo argumento es un fichero entonces sobreescribirá su contenido. Si se usa con más de un argumento el último ha de ser obligatoriamente un directorio.

```
$ cp fich1 fichdest
$ cp fich1 dirdest
$ cp fich1 fich2 fich3 dirdest
```

---

cpio(1)

Admite una lista de nombres de ficheros para empaquetarlos en un solo fichero. Es un comando muy potente y versátil. Junto con tar se usa entre otras cosas para hacer copias de seguridad. Suele usarse mucho en combinación con find. Lo estudiaremos más adelante.

---

cut(1)

Permite extraer columnas o campos desde uno o más ficheros.

```
$ cut -d: -f1 /etc/passwd
$ cut -d: -f6 /etc/passwd
```

---

date(1)

Obtiene la fecha. Las opciones permiten obtener distintos formatos de salida.

```
$ date

dom jun 11 18:17:14 CEST 2000

$ date +"%Y/%m/%d %T"

2000/06/11 18:16:49

$ date +%s

960740283
```

---

df(1)

Informa de la utilización de disco en las particiones que están montadas.

```
$ df -a
```

---

diff(1)

Sirve para localizar diferencias entre dos ficheros de texto.

```
$ diff fich1 fich2
```

---

du(1)

Permite averiguar la ocupación de espacio de todo aquello que cuelga de un determinado directorio. Este comando conviene usarlo de vez en cuando para localizar directorios demasiado cargados de información.

```
$ du -s .

$ du .

$ du -s * | sort -n
```

---

echo(1)

Permite sacar mensajes por salida estándar.

---

egrep(1)

Es una variedad del comando grep que permite el uso de expresiones regulares extendidas. Sirven para buscar cadenas de texto o secuencias de caracteres en ficheros. Se estudiarán más adelante.

---

emacs(1)

Es un editor multipropósito.

---

env(1)

Obtiene la lista de variables de entorno y sus valores.

---

ex(1)

Es un editor interactivo similar al vi. En Linux existe una versión mejorada llamada elvis.

---

expr(1) Es un evaluador de expresiones.

```
$ expr \( 55 + 31 \) / 3

28
```

---

false(1)

Solo tiene sentido usarlo en programación y retorna un código de retorno que indica error. En otras palabras no solo no hace nada sino que además siempre lo hace mal. Parece absurdo pero tiene su utilidad. Si no fuera sí no existiría y no es el único comando de este tipo.

---

fgrep(1)

Es una variedad del comando grep. Sirven para buscar cadenas de texto o secuencias de caracteres en ficheros. Se estudiarán más adelante.

---

file(1)

Sirve para averiguar el tipo de fichero pasado como argumento. Distingue muchos tipos de ficheros.

```
$ file fich
$ file dir1 fich2
```

---

find(1)

Permite localizar ficheros dentro de la estructura de directorios. Es tremendamente versátil y se estudiará más adelante.

```
$ find /var -name '*.log'
```

---

finger(1)

Sirve para averiguar quien está conectado al sistema.

```
$ finger
$ finger -l
```

---

free(1)

Proporciona estadísticas de uso de la memoria del sistema.

ftp(1)

Comando para intercambiar ficheros entre distintos ordenadores.

---

fuser(1)

Indica que proceso está usando un fichero o un directorio.

```
$ fuser /
```

---

gawk(1)

Es la versión GNU de awk. Se trata de un programa que implementa un lenguaje para tratar ficheros de texto estructurados en campos y registros. Es muy potente y se estudiará más adelante.

---

grep(1)

Junto a egrep, fgrep y rgrep sirven para buscar cadenas de texto o secuencias de caracteres en ficheros. Se estudiarán más adelante.

---

gzip(1)

Compresor des-compresor de gnu. Se puede usar directamente sobre un fichero o por el contrario puede usarse como filtro. Cuando se usa directamente sobre un fichero se modificará el nombre del fichero añadiendo .gz para indicar que está comprimido o se eliminará cuando sea descoprimido.

```
$ ls -l / > directorio_raiz
$ gzip directorio_raiz
$ gzip -d directorio_raiz.gz
$ gzip < directorio_raiz > directorio_raiz.gz
$ gzip -d < directorio_raiz.gz > directorio_raiz
```

---

head(1)

Permite sacar las primeras lineas de un fichero.

---

hostname(1)

Obtiene el nombre de la máquina.

---

id(1)

Devuelve información de identidad de aquel que lo ejecuta.

---

info(1)

Es un lector de hipertexto de GNU. Muchos programas de GNU incluyen documentación en formato info. Los hipertextos son textos que incluyen enlaces sub-menús, y otros elementos que facilitan una lectura no secuencial. Por ejemplo el html es otro formato de hipertexto.

---

ispell(1)

Es un comprobador ortográfico que puede ser acoplado a diversos editores como por ejemplo al vi. Existen diccionarios ortográficos para distintos lenguajes.

---

kill(1)

Envía una señal a un proceso. Se llama kill porque generalmente se usa para "matar" procesos. Requiere que se conozca el identificador del proceso PID.

---

killall(1)

Como el anterior pero permite pasar el nombre del comando y "matará" a todos los procesos que tengan ese nombre.

---

less(1)

Es un paginador que puede usarse en sustitución del paginador more.

```
$ less fichero
$ cat fichero | less
```

ln(1)

Crea enlaces rígidos y enlaces simbólicos que son cosas que ya hemos estudiado.

---

locate(1)

Permite la localización rápida de ficheros en su sistema de ficheros. Se utiliza una base de datos que debe de ser actualizada regularmente mediante el comando updatedb. Este último lo realizará root cuando lo crea oportuno o mejor aun estará programado en el sistema para que se ejecute periódicamente. La idea es la siguiente. Para localizar un fichero se suele ejecutar el comando find capaz de explorar todo el árbol del sistema de ficheros pero esta operación es muy costosa porque requiere abrir y cerrar cada uno de los directorios en los que busca información. El comando updatedb hace esto mismo pero guardando el resultado de toda la exploración en una base de datos muy compacta donde se puede buscar con enorme rapidez. La desventaja es que la información de búsqueda tendrá cierta antigüedad y que no se guardan todos los atributos del fichero por lo que resulta imposible buscar usando criterios para esos atributos. Por ello usaremos locate para ciertas búsquedas aunque algunas veces tendremos que usar find.

---

lpq(1)

Informa sobre el estado de la cola de impresión. Muestra las tareas que están el la cola su identificador numérico, orden, tamaño y propietario.

---

lprm(1)

Permite cancelar tareas de impresión usando el identificador numérico obtenido con lpq..

---

ls(1)

Obtiene un listado de los ficheros de un directorio. Tiene una gran cantidad de opciones.

---

mail(1)

Se usa para enviar un correo electrónico. Su manejo comparado con otros programas de correo electrónico no resulta muy amigable pero no solo sirve para usarlo en modo interactivo. Se le puede pasar por la entrada estándar el contenido de un mensaje. Una aplicación de esto es que podemos coleccionar mensajes cortos por ejemplo para suscribirse o de-suscribirse de listas de correo.

```
$ echo unsubscribe l-linux | mail majordomo@calvo.teleco.ulpgc.es
$ mail -s unsubscribe debian-user-spanish-request@lists.debian.orga < null>
```

man(1)

Este fué el primer comando que se estudió en este curso y no nos cansaremos de recomendarle que se familiarice con el. La información de estas páginas del manual no suele resultar muy didáctica. En Linux no vienen apenas ejemplos. A pesar de ello suele traer información completa de opciones y formas de uso.

---

mkdir(1)

Crea uno o más directorios. Los permisos dependerán de el valor actual de 'umask'. Esto ya lo vimos en capítulos anteriores. Para eliminar un directorio se usará 'rmdir'.

```
$ mkdir dir1 dirA/dirB/dirC/dir2 dir3
```

---

more(1)

Es un paginador mucho más sencillo que el sofisticado 'less'.

```
$ ls | more

$ more fichero
```

---

mv(1)

Se utiliza para renombrar directorios y ficheros o para trasladarlos de un lugar a otro. Conviene usarlo con precaución porque se presta a obtener resultados diferentes en función de la existencia o no de un fichero o directorio de destino. Mover un fichero a otro que ya existe supondría sobreescribirlo así que un error puede provocar perdida de información. Cuando se usan más de dos argumentos en Linux se obliga a que el último sea un directorio de destino lo cual previene errores.

```
$ mv fich1 fich01
$ mv dir1 dir01
$ mv fich1 fich2 dir1 dir2 fich3 fich4  dir001
```

nice(1)

A un usuario normal le permitirá bajar la prioridad de los proceso lanzados con este comando. Solo root puede usarlo para aumentar la prioridad. Una vez que un comando ha sido lanzado se puede modificar su prioridad con renice.

```
$ nice comando
```

---

nohup(1)

Los comandos lanzados con nohup no terminan al abandonar la sesión. Se puede usar combinado con la ejecución en background.

```
$ nohup GeneraInformeMensual > informe01-05-2000.out &

$ # logout
$ exit
```

---

passwd(1)

Permite cambiar nuestra clave de acceso al sistema. Conviene no olvidar la clave. Generalmente para evitar olvidos los usuarios eligen claves demasiado sencillas. Dependiendo del nivel de seguridad configurado en el sistema, este podría rechazar claves demasiado sencillas. No conviene usar claves cortas ni palabras que puedan ser localizadas en un diccionario ni fechas o nombres relacionadas con datos de alto significado personal. Palabras escritas al revés tampoco son seguras. Intercalar algún carácter de puntuación alguna mayúscula o algún dígito suele ser una buena práctica. foTo;21 pon.5.mar 7li-bRos Bueno si el ordenador es de uso personal y solo tienen acceso personas de confianza tampoco hay que ser demasiado paranoico.

---

paste(1)

Se usa para combinar columnas de distintos ficheros en uno solo. Viene a ser lo contrario del comando cut.

```
$ who | paste - -
```

---

pr(1)

Permite paginar un texto incorporando cabeceras.

```
$ ls /*/* | pr -h ejemplo-pr -o 5 -l 35 | less
```

---

ps(1)

Permite obtener información de los procesos en ejecución. Dependiendo lo que nos interese existen diversas opciones para una gran variedad de formatos y de selección de la información de salida.

---

pstree(1)

Muestra la jerarquía entre procesos mostrando las relaciones de parentesco entre ellos.

```
$ pstree

$ pstree -p
```

La opción -p es muy útil para averiguar el pid de un proceso.

---

pwd(1)

Este comando retorna el lugar donde nos encontramos.

```
$ pwd
```

Se puede usar por ejemplo para guardar el lugar donde estamos ahora con objeto de retornar ala mismo lugar más tarde.

```
$ AQUI=`pwd`
$ cd /tmp
$ ls
$ cd $AQUI
```

---

renice(1)

Es parecido a nice pero actúa sobre un proceso que ya fue arrancado con anterioridad. Por ejemplo hemos lanzado un proceso que consume muchos recursos y lleva mucho tiempo. En lugar de pararlo podemos bajarle la prioridad para que gaste muchos recursos. (Cuando decimos recursos nos referimos a uso de memoria, uso de CPU, etc) Un super usuario como root puede incrementar la prioridad usando un número negativo. Un usuario normal solo podrá decrementar la prioridad usando un número positivo. En cualquier caso se requiere conocer el pid del proceso que deseamos modificar y solo podremos hacerlo si es un proceso nuestro. Por ejemplo vamos a suponer que el PID de ese proceso que deseamos bajar de prioridad es el 778.

```
$ renice +15 778
```

---

reset(1)

Puede ocurrir que el terminal quede des-configurado por alguna razón. Esto se debe a que un terminal interpreta comandos en forma de secuencias de caracteres. Esto permite mostrar colores manejar el cursor y una serie de cosas más. Cuando accidentalmente enviamos al terminal un fichero binario que contiene caracteres de todo tipo en cualquier orden, es bastante normal que el terminal quede inutilizable. Muchas veces se puede recuperar introduciendo el comando reset.

---

rm(1)

Ya hemos explicado y usado este comando. Sirve para borrar ficheros.

---

rmdir(1)

Ya hemos explicado y usado este comando. Sirve para eliminar directorios que deben de estar vacíos.

---

rsh(1)

Permite siempre que tengamos permiso ejecutar un comando en un ordenador distinto. Se puede usar para transferir grandes cantidades de información a través de la red. Por ejemplo imaginemos que queremos sacar una copia de seguridad guardando la información en una unidad de cinta que se encuentra en un ordenador distinto. En ese caso se lanzan dos comandos simultáneos comunicándolos con un pipe. Un comando se ejecutará en el ordenador remoto con rsh y el otro se ejecuta en local. Ambos procesos quedan comunicados por entrada salida pero eso implica que esa comunicación viajará a través de la red.

```
$ ### Para salvar
$ tar cf - . | rsh remotehost dd of=/dev/tape

$ ### Para recuperar
$ rsh remotehost dd if=/dev/tape | tar xf -
```

La forma de combinar comandos en unix conectando entrada salida permite hacer muchas veces cosas sorprendentes como en este caso. Existen un comando similar que hace lo mismo pero la información viaja encriptada a través de la red. Se trata de 'ssh'.

script(1)

Este es un comando realmente curioso. Su utilidad principal es grabar una sesión. Lanza una subshell que se ejecutara en un pseudo-terminal. Este palabro no deseamos explicarlo ahora pero mientras estamos en esa subshell todos los caracteres recibidos por el terminal son grabados en un fichero. Para terminar se teclea exit. Es útil para memorizar sesiones de trabajo.

---

sed(1)

Este es un editor de flujo. Dicho de otra forma es un editor que está pensado para usarlo como flujo. De esta forma se puede usar como una poderosa herramienta para transformar texto. Se estudiará más adelante.

---

sleep(1)

Sirve para esperar sin consumir recursos. El proceso queda dormido durante el número de segundos indicado.

```
$ echo hola; sleep 5 ; echo que tal
```

---

sort(1)

Este comando permite ordenar lineas de texto. Se puede usar como filtro. Se estudiará más adelante.

---

split(1)

Este comando sirve para partir en trozos más pequeños un fichero grande. Para volver a obtener el fichero original bastará con usar el comando 'cat'

```
$ split --bytes=52m ficherogrande parte
```

Este comando trocea ficherogrande en trozos de 52Mbytes que se guardan en ficheros que empiezan con el prefijo 'parte' seguido de una numeración secuencial.

stty(1)

Sirve para comprobar el estado actual del terminal y para cambiar la configuración del mismo. Se puede cambiar el significado de algunos caracteres de control, establecer el número de filas y columnas del terminal, la velocidad de transmisión, etc. Para comprobar el estado actual teclee lo siguiente.

```
$ stty -a
```

---

su(1)

Permite cambiar de usuario. Lo que se hace es lanzar una subshell. Hay dos formas de hacerlo. 'su nuevousuario' y 'su - nuevousuario'. Si no usamos la segunda forma solo cambiará el usuario efectivo pero el entorno del proceso se mantiene. Si por el contrario se usa la segunda forma se ejecutarán los scripts de inicio de sesión para el nuevo usuario y tendremos una sesión idéntica a la obtenida con la entrada con login para ese nuevo usuario. Para terminar esa sesión bastará hacer exit.

```
$ su nuevousuario

$ su - nuevousuario
```

Para un usuario normal se solicitará la password del nuevo usuario.

---

tail(1)

Así como head servía para obtener las primeras líneas de un fichero tail sirve para obtener las últimas líneas de un fichero

```
$ tail fichero
```

Tail tiene una opción muy útil para mostrar que es lo que está pasando en ficheros que se están generando o que crecen continuamente.

```
$ tail -f fichero
```

Este comando no termina nunca y muestra el final del fichero quedando a la espera de mostrar el resto de mismo a medida que se genere. Para terminar hay que matarlo con Ctrl-C.

---

talk(1)

Permite abrir una sesión de charla interactiva con otro usuario.

---

tee(1)

Este comando se utiliza para obtener una bifurcación en un flujo de entrada salida. Actúa como una derivación en forma de 'T'.

```
$ ll | tee todos-fich | tail > ultimos-fich
```

---

telnet(1)

Permite abrir una sesión de trabajo en otra máquina.

```
$ telnet localhost
```

Con esto podemos probar telnet conectándonos con nuestra propia máquina. A continuación saldrá la solicitud de login.

---

test(1)

Se usa para evaluar expresiones. Se usa mucho en la programación shell-script. Lo que se usa es su código de retorno. Tiene dos formas de uso 'test expresión' o '[expresión]'

```
$ test "2" = "3"
$ echo $?

$ test "2" = "2"
$ echo $?

$ A=335
$ B=335

$ [ "$A" = "$B" ]
$ echo $?
```

Se estudiará más adelante.

tload(1)

Convierte el terminal en un monitor semi-gráfico que indicará el nivel de carga del sistema. Se interrumpe con Ctrl-C. Si el sistema no está trabajando demasiado no verá nada demasiado interesante.

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

$ tload -d 2
```

---

top(1)

Muestra información de cabecera con estadísticas de uso de recursos. (número de procesos y en que estado están, consumo de CPU, de memoria, y de swap). Además muestra información muy completa sobre los procesos que están consumiendo más recursos. Si el ordenador va muy lento por exceso de trabajo podemos hacernos una idea muy buena de los motivos usando este comando. En realidad este comando es muy útil para el administrador del sistema pero los usuarios que comparten el uso de un sistema con otros usuarios también tienen que usar los recursos del sistema de forma inteligente bajando la prioridad de tareas poco urgentes y que consuman mucho. En especial no resulta buena idea lanzar muchas tareas pesadas de forma simultanea aunque sean urgentes porque se perjudica el rendimiento global del sistema que gastará excesivo tiempo en labores improductivas de gestión de procesos.

```
$ top
```

Para salir hay que pulsar 'q'.

---

tr(1)

Se utiliza para sustituir carácter.

```
$ # Pasar a mayúsculas
$ tr '[a-z]' '[A-Z]' < fichero > nuevofichero

$ # Eliminar el carácter ':'
$ tr -d : < fichero > nuevofichero
```

---

true(1)

Este comando es el contrario del comando 'false'. Solo tiene sentido usarlo en programación y retorna un código de retorno que indica éxito. En otras palabras no hace pero al menos lo hace correctamente.

vi(1)

Este es un editor muy potente y presente en todos los sistemas de tipo Unix. Se estudiará más adelante.

---

w(1)

Muestra información de usuarios actualmente conectados mostrando momento de inicio de sesión y el consumo de CPU que se ha realizado.

```
$ w
```

---

wc(1)

Este es un comando que se usa bastante. Sirve para contar caracteres, palabras y lineas en un fichero o en un flujo de entrada salida.

```
$ wc fichero

$ cat fichero | wc
```

---

whatis(1)

Sirve para mostrar la breve descripción de un comando que figura en las páginas del manual

```
$ whatis ls

$ whatis whatis
```

---

who(1)

Saca la información de quienes están conectados al sistema.

```
$ who
```

---

whoami(1)

Para averiguar quien es usted en ese momento. En un programa puede ser interesante comprobar quien lo está ejecutando.

```
$ whoami
```

---

whereis(1)

Sirve para localizar un comando en el sistema siempre que este esté localizado en $PATH

```
$ whereis ls

$ whereis whereis
```

---

write(1)

Siempre que se tenga permiso permite enviar mensajes a otro usuario.

```
$ write root
```

---

xargs(1)

Sirve para pasar argumentos a un comando mediante entrada salida.

```
$ echo '-l' | xargs ls
```

Cosas como estas añaden bastante potencia al lenguaje shell-script.

---

z...

Hay una serie de comandos. 'zcat, zmore, zgrep, zless, zdiff' que permiten trabajar directamente con ficheros comprimidos con gzip en la misma forma que lo haríamos directamente con 'cat, more, grep, less y diff' respectivamente.

```
$ zless documento.txt.gz
```

EXPRESIONES REGULARES

- Introducción.
- Operadores usados en expresiones regulares.
- Ejemplos para cada operador con 'egrep'.
- Ampliando conocimientos sobre 'egrep'.
- Uso de expresiones regulares en 'sed'.
- Ampliando conocimientos sobre 'sed'.
- Test

Introducción

Vamos a explicar las expresiones regulares porque se utilizan a menudo desde una gran variedad de aplicaciones en los SO tipo Unix como Linux. Permiten reconocer una serie de cadenas de caracteres que obedecen a cierto patrón que llamamos expresión regular. Por ejemplo si deseamos buscar lineas que contenga las palabras 'hola' o 'adiós' en los ficheros del directorio actual haremos:

```
$ egrep 'hola|adiós' *
```

No todos los comandos usan de forma idéntica las expresiones regulares. Algunos de los comandos que usan expresiones regulares son 'grep', 'egrep', 'sed', 'vi', y 'lex'. Este último en linux se llama 'flex' y es un analizador sintáctico muy potente pero no lo explicaremos porque para usarlo hay que saber lenguaje 'C'. Actualmente algunos lenguajes modernos como el 'perl' incluyen capacidad de manejar expresiones regulares lo cual les da una gran potencia y para lenguajes más antiguos como el 'C' existen librerías para poder usar expresiones regulares. En resumen las expresiones regulares están sinedo incorporadas en distintos sitios y ya no están limitadas a su uso en SO tipo Unix. Cada comando o aplicación implementa la expresiones regulares a su manera aunque en general son todas bastantes parecidas. Por ejemplo 'grep' permite usar expresiones regulares sencillas mientras que 'egrep' es capaz de usar expresiones regulares más complejas. Otros comandos adaptan el uso de expresiones a sus particulares necesidades y por ello si bien se puede hablar en general de ellas hay que tener en cuenta las peculiaridades de cada caso que deberán ser consultadas en las paginas del manual de cada comando. Las expresiones regulares vienen a ser una especie de lenguaje y cada comando usa su propio dialecto. En realidad las diferencias entre los distintos dialectos suelen ser muy pocas. Por ejemplo si un comando usa los paréntesis y además admite el uso de expresiones regulares extendidas se establecerá una forma de distinguir si los paréntesis deben ser interpretados como patrón de la expresión regular o como otra cosa. Para ello se suele usar los paréntesis precedidos del carácter escape '\'. Vamos a tomar a 'egrep' y 'sed' como comandos para aprender expresiones regulares porque este curso tiene un enfoque práctico. Usaremos el comando 'egrep' con distintos patrones y veremos cuando cumple y cuando no cumple y de esa forma se entenderá perfectamente.

Operadores usados en expresiones regulares.

*****	El elemento precedente debe aparecer 0 o más veces.
+	El elemento precedente debe aparecer 1 o más veces.
.	Un carácter cualquiera excepto salto de linea.
?	Operador unario. El elemento precedente es opcional

|	O uno u otro.
^	Comienzo de linea
$	Fin de linea
[...]	Conjunto de caracteres admitidos.
[^...]	Conjunto de caracteres no admitidos.
-	Operador de rango
(...)	Agrupación.
****	Escape
\n	Representación del carácter fin de linea.
\t	Representación del carácter de tabulación.

Esta lista no es completa pero con esto es suficiente para hacer casi todo lo que normalmente se hace con expresiones regulares.

Ejemplos para cada operador con 'egrep'
Empezaremos usando un ejemplo lo más sencillo posible para ilustrar cada uno de estos operadores

Ejemplo para el operador '*' con el patrón 'ab*c'
Este patrón localizará las cadenas de caracteres que empiecen por 'a', que continúen con 0 o más 'b', y que sigan con una 'c'.

La 4 falla porque la 'a' y la 'c' no van seguidas ni existen caracteres 'b' entre ambas. La 5 falla por no tener una sola 'c'. y la 6 tiene los caracteres adecuados pero no en el orden correcto.

```
$ egrep 'ab*c' <<FIN
< 1 ac
< 2 aac
< 3 abbbc
< 4 axc
< 5 aaab
< 6 cba
< 7 aacaa
< FIN

1 ac
2 aac
3 abbbc
7 aacaa
```

Ejemplo para el operador '+' con el patrón 'ab+c'
Este patrón localizará las cadenas de caracteres que empiecen por 'a', que continúen con 1 o más 'b', y que sigan con una 'c'.

Solo la línea 3 cumple la expresión regular.

```
$ egrep 'ab*c' <<FIN
< 1 ac
< 2 aac
< 3 abbbc
< 4 axc
< 5 aaab
< 6 cba
< 7 aacaa
< FIN

3 abbbc
```

Ejemplo para el operador '.' con el patrón 'a..c'
Este patrón localizará las cadenas de caracteres que empiecen por 'a', que continúen con dos caracteres cualesquiera distintos de salto de línea y que sigan con una 'c'.

```
$ egrep 'a..c' <<FIN
< a00c
< axxcxx
< aacc
< abc
< ac
< axxxc
< FIN

a00c
axxcxx
aacc
```

Ejemplo para el operador '?' con el patrón 'ab?cd'
Este patrón localizará las cadenas de caracteres que empiecen por 'a', y que opcionalmente continúen con una 'b' y que continúe con los caracteres 'cd'.

```
$ egrep 'ab?cd' <<FIN
< abcd
< acd
< cd
< abbcd
< FIN

abcd
acd
```

Ejemplo para el operador '|' con el patrón 'ab|cd'
Este patrón localizará las cadenas de caracteres que contengan 'ab', 'cd', o '123'

```
$ egrep 'ab|cd|123' <<FIN
< xxxabzzz
< xxxcdkkk
< badc
< dbca
< x123z
< FIN

xxxabzzz
xxxcdkkk
x123z
```

Ejemplo para el operador '^' con el patrón '^abc'
Este patrón localizará las lineas que empiecen por 'abc'.

```
$ egrep '^abc' <<FIN
< abcdefgh
< abc xx
< 00abc
< FIN

abcdefgh
abc xx
```

Ejemplo para el operador '$' con el patrón 'abc$'
Este patrón localizará las lineas que terminen por 'abc'.

```
$ egrep 'abc$' <<FIN
< abcd
< 000abc
< xxabcx
< abc
< FIN

000abc
```

```
abc
```

Ejemplo para el operador '[ ]' con el patrón '0[abc]0'
Este patrón localizará las cadenas que tengan un '0' seguido de un carácter que puede ser 'a', 'b', o 'c' y seguido de otro '0'.

```
$ egrep '0[abc]0' <<FIN
< 0abc0
< 0a0
< 0b0
< 0c0
< xax
< FIN

0a0
0b0
0c0
```

Ejemplo para el operador '^' (negación) dentro de '[ ]' con el patrón '0[^abc]0'
Este patrón localizará las cadenas que tengan un '0' seguido de un carácter que no podrá ser ni 'a', ni 'b', ni 'c' y seguido de otro '0'.

```
$ egrep '0[^abc]0' <<FIN
< 0abc0
< 0a0
< 000
< x0x0x
< 0c0
< FIN

000
x0x0x
```

Ejemplo para el operador '-' (rango) dentro de '[ ]' con el patrón '0[a-z]0'
Este patrón localizará las cadenas que tengan un '0' seguido de una letra minúscula, y seguido de otro '0'.

```
$ egrep '0[a-z]0' <<FIN
< 0a0
< 000
< 0Z0
< x0x0x
< FIN

0a0
x0x0x
```

Ejemplo para el operador '()' (agrupación) con el patrón '0(abc)?0'
Este patrón localizará las cadenas que tengan un '0' seguido opcionalmente de la secuencia abc, y seguido de otro '0'.

```
$ egrep  '0(abc)?0' <<FIN
< hh00hh
< s0abc0s
< s0ab0s
< 0abc
< FIN
hh00hh
s0abc0s
```

Ejemplo para el operador '\' (escape) con el patrón '0\(abc\)?0'
Este patrón localizará las cadenas que tengan un '0' seguido opcionalmente de ')', y seguido de otro '0'.

```
$ egrep  '0\(abc\)?0' <<FIN
< 0(abc)0xx
< 0(abc0xx
< hh00hh
< FIN
0(abc)0xx
0(abc0xx
```

Ampliando conocimientos sobre 'egrep'
Hemos aprendido cosas de egrep para ilustrar el uso de las expresiones regulares pero tanto grep como egrep permiten el uso de ciertas opciones que aún no hemos comentado y que no vamos a ilustrar con ejemplos porque no están relacionadas con las expresiones regulares. Solo vamos a señalar las opciones más útiles para que las conozca.

- -i Busca ignorando diferencias entre mayúsculas y minúsculas.
- -w Para forzar que la cadena reconocida sea una palabra completa.
- -l No muestra el contenido de la linea encontrada pero si que muestra el fichero que contiene la cadena buscada. Esto es útil cuando deseamos buscar entre muchos ficheros de los cuales algunos son binarios porque la muestra del contenido de un fichero binario puede desconfigurar el terminal.
- -n Muestra el número de línea dentro del fichero para ayudar a su localización.
- -v En lugar de sacar la lineas que cumplen la búsqueda sacará las que no cumplen.

Para un mayor detalle consultar las páginas del manual. Se trata de un comando de gran utilidad y le recomendamos que practique por su cuenta con el. Exísten varias modalidades de este comando.

- grep Comando básico.
- egrep Versión grep para uso de expresiones regulares extendidas.
- fgrep Versión grep interpreta los patrones no como expresiones regulares sino como cadenas de caracteres fijas.
- rgrep Permite buscar recursivamente en varios ficheros recorriendo un arbol de directorios.

Uso de expresiones regulares en 'sed'
Ahora veremos unos ejemplos con 'sed'. Este comando es capaz de editar un flujo o chorro de caracteres lo cual es de enorme utilidad dado que muchos comandos se comunican a través de entrada salida mediante chorros de caracteres. No podemos ver todas las posibilidades de 'sed' porque lo que nos interesa es su uso con expresiones regulares. Para ello usaremos un nuevo operador '&' que en la parte de substitución de 'sed' indica la parte que coincide con la expresión regular.

Para usar 'sed' se pone como primer argumento a continuación del comando la orden adecuada. Opcionalmente se puede poner un segundo parámetro que indicaría el nombre de un fichero. De no existir ese segundo parámetro esperará la entrada por la entrada estándar.

Se intentan reconocer las secuencias más largas posibles ^ y $ no consumen carácter '\n' si.

Empezamos con una sustitución sencillita.

```
$ echo "abc1234def" | sed "s/[0-9][0-9]*/NUMERO/"

abcNUMEROdef
```

Ahora usamos el operador '&'. Observe como se sustituye por el patrón reconocido.

```
$ echo "abc1234def" | sed "s/[0-9][0-9]*/<&>/"

abc<1234>def
```

Después eliminamos la secuencia numérica.

```
$ echo "abc1234def" | sed "s/[0-9][0-9]*//"

abcdef
```

Vamos a comprobar que en las expresiones regulares se intenta siempre reconocer la secuencia más larga posible.

```
$ echo "000x111x222x333" | sed "s/x.*x/<&>/"

000<x111x222x>333
```

Vamos ahora a trabajar sobre un fichero. Como siempre recordamos que trabaje con un usuario sin privilegios y dentro de /tmp. Ya sabe como hacerlo así que cambie ahora a /tmp antes de continuar. Para suprimir del fichero la palabra 'Hola' en las lineas de la 3 a la 4.

```
# Creamos un ficherito de prueba
$ cat <<FIN > prueba-sed.txt
< Hola este es un fichero con datos de prueba
< Hola otra vez.
< Hola otra vez.
< Hola otra vez.
< Hola otra vez y otra y otra y otra y otra y otra.
< Fin de los datos de prueba
< FIN
$ sed "3,4s/Hola//" prueba-sed.txt

Hola este es un fichero con datos de prueba
Hola otra vez.
 otra vez.
 otra vez.
Hola otra vez y otra y otra y otra y otra y otra.
Fin de los datos de prueba
```

El fichero no ha cambiado. El resultado sale por salida estándar. Ahora veamos que pasa si intentamos sustituir la palabra 'otra' por la palabra 'una' en todo el fichero.

```
$ sed "s/otra/una/" prueba-sed.txt

Hola este es un fichero con datos de prueba
Hola una vez.
Hola una vez.
Hola una vez.
Hola una vez y otra y otra y otra y otra y otra.
Fin de los datos de prueba
```

Vemos que solo se ha cambiado la primera ocurrencia de cada línea. Para obtener el resultado deseado tendríamos que usar la g al final.

```
$ sed "s/otra/una/g" prueba-sed.txt

Hola este es un fichero con datos de prueba
Hola una vez.
Hola una vez.
Hola una vez.
Hola una vez y una y una y una y una y una.
Fin de los datos de prueba
```

Ampliando conocimientos sobre 'sed'

Hemos aprendido cosas de sed para ilustrar el uso de las expresiones regulares pero sed tiene muchas más posibilidades que aún no hemos comentado. Comentaremos solo algo más pero sin extendernos demasiado. En sed se pueden especificar varias instrucciones separando con ';' cada una de ellas o usando la opción -e antes de cada instrucción. También podemos aprovechar el segundo introductor de la shell.

```
$ sed "s/otra/vez/g ; s/vez/pez/g" prueba-sed.txt

Hola este es un fichero con datos de prueba
Hola pez pez.
Hola pez pez.
Hola pez pez.
Hola pez pez y pez y pez y pez y pez y pez.
Fin de los datos de prueba

# Idéntico resultado podríamos haber conseguido usando
$ sed -e "s/otra/vez/g"  -e "s/vez/pez/g" prueba-sed.txt
# Una tercera forma para conseguir lo mismo
$ sed "
< s/otra/vez/g
< s/vez/pez/g
< " prueba-sed.txt
```

También podemos obtener el mismo resultado usando la opción -f y un fichero de instrucciones para 'sed'.

```
$ cat  <<FIN > prueba-sed.sed
< s/otra/vez/g
< s/vez/pez/g
FIN
$ sed -f prueba-sed.sed prueba-sed.txt
```

Existen posibilidades más avanzadas para usar 'sed' pero con lo que hemos mencionado se pueden hacer muchas cosas.

Podemos eliminar los blancos a principio y al final de linea así como sustituir mas de un blanco seguido por un solo blanco.

```
$ cat  <<FIN > trim.sed
< s/^  *//g
< s/  *$//g
< s/  */ /g
< FIN
```

Si necesita hacer alguna transformación y no encuentra la forma facil de hacer con sed piense que quizás necesite usar varias transformaciones en cadena.

```
#!/bin/bash
##########################################################
echo ; echo
# Filtraremos la salida del comando cal para que el día de
# hoy resalte enmarcada entre corchetes.
#
# Primero obtenemos el día del més
DIAMES=`date '+%d'`
# Asi no vale. Hay que eliminar los posibles ceros a la
# izquierda obtenidos # en la variable DIAMES.
# Usamos para ello printf(1)
DIAMES=`printf %d $DIAMES`
# Resaltar el día del més correspondiente al día de hoy,
# Si es el último día del mes (patrón ' [23][0-9]$') añadiremos
# un blanco al final y así usaremos siempre como patrón de
# sustitución el día més ($DIAMES) entre dos caracteres blancos
# sin que eso provoque errores en el último diá del mes.
cal | head -n 1
cal | tail -n 7 | sed 's/  / +/g' | sed 's/^ /+/g' | \
sed 's/ /  /g' | sed 's/+/ /g' | sed 's/$/& /' | \
sed 's/^/ /' | sed "s/ $DIAMES /\[${DIAMES}\]/"
# Trazas
cal | tee traza.1 | \
tail -n 7 | tee traza.2 | \
sed 's/  / +/g' | tee traza.3 | \
sed 's/^ /+/g' | tee traza.4 | \
sed 's/ /  /g' | tee traza.5 | \
sed 's/+/ /g' | tee traza.6 | \
sed 's/$/& /' | tee traza.7 | \
sed 's/^/ /' | tee traza.8 | \
sed "s/ $DIAMES /\[${DIAMES}\]/" > traza.9
```

Las últimas líneas las hemos añadido simplemente para analizar paso a paso como se va transformando la salida original del comando cal del cual en primer lugar ya hemos separado la primera línea. Las distintas etapas quedan registradas en traza.1 a traza.8.

```
traza.1 contiene la salida del comando cal.

     April 2002
 S  M Tu  W Th  F  S
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30

En traza.2 se ha eliminado la primera línea.

 S  M Tu  W Th  F  S
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
```

```
28 29 30
```

En traza.3 se ha sustituido toda secuencia de dos blancos por ' +'

```
 S +M Tu +W Th +F +S
 + +1 +2 +3 +4 +5 +6
 7 +8 +9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
```

En traza.4 sustituimos los blancos a comienzo de línea por '+'

```
+S +M Tu +W Th +F +S
++ +1 +2 +3 +4 +5 +6
+7 +8 +9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
```

Gracias a los pasos anteriores ya tenemos una estructura de líneas formada por dos caracteres distintos de blanco separadas por un blanco y precisamente este blanco en traza.5 lo sustituimos por dos blanco para dar ensanchar toda la salida.

```
+S  +M  Tu  +W  Th  +F  +S
++  +1  +2  +3  +4  +5  +6
+7  +8  +9  10  11  12  13
14  15  16  17  18  19  20
21  22  23  24  25  26  27
28  29  30
```

Ahora ya no son útiles los '+' y los sustituimos por blancos. En traza.6 ya tenemos la salida ensanchada.

```
 S   M  Tu   W  Th   F   S
     1   2   3   4   5   6
 7   8   9  10  11  12  13
14  15  16  17  18  19  20
21  22  23  24  25  26  27
28  29  30
```

En traza.7 no se aprecia la diferencia pero si que existe. cal termina las líneas inmmediatamente despues del número, y no hay razón para que lo haga de otra manera pero a nosotros nos conviene que absolutamente todos los números tengan un blanco a la izquierda y otro a la derecha para que a la hora de enmarcar el número solo tengamos que sustituir blancos y no se desalineen las columnas. Por ello aunque no se aprecie en traza.7 hemos añadido un blanco al final de las líneas.

```
 S   M  Tu   W  Th   F   S
     1   2   3   4   5   6
 7   8   9  10  11  12  13
14  15  16  17  18  19  20
21  22  23  24  25  26  27
28  29  30

Ahora en traza.8 se ve como hemos añadido un
blanco a la izquierda.

  S   M  Tu   W  Th   F   S
      1   2   3   4   5   6
  7   8   9  10  11  12  13
 14  15  16  17  18  19  20
 21  22  23  24  25  26  27
 28  29  30

Por último bastará sustituir el día del més
entre blancos por el día de més entre corchetes.

  S   M  Tu   W  Th   F   S
      1   2   3   4   5   6
  7   8   9  10  11  12  13
 14  15  16  17  18  19 [20]
 21  22  23  24  25  26  27
 28  29  30
```

Independientemente de que a usted se le ocurra otra solución mejor, se trataba de practicar con sed y con siete seds encadenados pensamos que ya habrá saciado su sed de sed(1).

De todas formas si lo quiere más bonito y cortito pruebe esto otro.

```
ESC=`echo -e "\033"`
cal | sed "s/$/& /" | sed "s/ $DIAMES / $ESC[1m${DIAMES}$ESC[0m /"
```

Ya nos salimos del tema pero si esto ha despertado su curiosidad mire console_codes(4)

EL EDITOR VI (primera parte)

- Introducción.
- 'ed' versus 'vi'.
- 'vim' versus 'vi'.
- Cuadro resumen para el uso de 'vi'.
- Practicando lo más esencial de 'vi'.
- Recuperar la información perdida por muerte prematura del editor.
- Test

Introducción

En primer lugar vamos a dedicar bastante esfuerzo a explicar porque aprender 'vi'. Quizás pensó que este capítulo no era para usted porque ya sabe usar otro editor. También vamos explicar la necesidad de dedicar dos largos capítulos a este editor.

Cuando finalice este capítulo podrá editar sus ficheros con 'vi' y hacer bastante trabajo útil pero para ello quizas necesite algo más de esfuerzo que en capítulos anteriores y no porque sea dificil sino porque tendrá que practicar bastante por su propia cuenta.

Otra cosa será sacarle el máximo partido a 'vi'. En un segundo capítulo dedicado a 'vi' veremos opciones más avanzadas. Son opciones que quizás no necesite usar tan frecuentemente pero cuando tenga oportunidad de usarlas comprenderá la potencia el editor 'vi'.

Muchos editores actuales permiten usar distintos tipos de letras y generan documentos con un aspecto precioso que además coincide lo que se ve en pantalla cuando se edita, y con lo que saldrá por la impresora. Está claro que 'vi' no sirve para esto. 'vi' está diseñado para editar ficheros en formato de texto estándar. Pero aunque solo sirva para este formato de texto lo hace muy bien y siempre tendremos oportunidad de usarlo porque a un compilador, o a un interprete le da igual la estética.

Son muchos los editores disponibles en Linux pero desde un principio dijimos que este curso podría servir en gran medida para aprender otros SO tipo Unix. El editor 'vi' está presente desde hace mucho tiempo en un montón de sistemas tipo Unix. Aquí explicaremos solo unos pocos aspectos de 'vi' con la idea de que pueda realizar con razonable soltura la edición de un texto en 'vi'. Con muy pocos comandos de 'vi' se pueden hacer muchas cosas y 'vi' es un editor que para ciertas tareas resulta de una potencia increíble. El que escribe estas lineas lo está haciendo con 'vi' y lleva muchos años usando este editor prácticamente para casi todo y le saco un gran partido, pero cada vez que me encuentro con alguién que sabe bastante de 'vi' aprendo algo nuevo. 'vi' es un editor muy práctico sin tener que conocerlo a fondo pero en manos de un experto resulta de una eficacia espectacular.

'vi' es un editor de lineas interactivo y ha sido muy criticado. Una de las razones para ello es que tiene varios estados o formas de funcionamiento y si uno pasa de un estado a otro sin darse cuenta y continua tecleando cosas, puede ocurrir un pequeño destrozo en el fichero que está editando. Otro inconveniente de 'vi' es que requiere un poquito de aprendizaje. Hay muchísima gente que está acostumbrada a usar otros editores más intuitivos y porque no decirlo más cómodos, y rechazan a 'vi' como algo prehistórico. Otra cosa que suele criticarse es que se usan letras del teclado para cosas tan normales como el movimiento del cursor y la paginación del texto pero este es un claro ejemplo del condicionamiento de la cultura de Microsoft y de los PCs. 'vi' fue diseñado para que se pudiera usar en una gran variedad de sistemas y terminales. El uso de las teclas de flechas puede lograrse mediante la adecuada configuración de 'vi'. Uno de los puntos fuertes de 'vi' es su adaptabilidad.

Quizás una de las más poderosas razones para aprender 'vi' es que 'vi' forma parte de la cultura de los SO tipo Unix. Muchas personas acostumbradas a trabajar por ejemplo con Windows asumirán como estándar cosas que no lo son y cuando se encuentran algo diferente intentan aprovechar al máximo sus conocimientos anteriores.

Generalmente existe una exageradísima tendencia a continuar usando el primer editor que se aprendió y luego se cambia solo de editor obligado por circunstancias poderosas. Nosotros hemos decidido que en posteriores lecciones asumiremos que usted ya sabe usar 'vi'. Pronto tendrá que hacer sus primeros programas en shell-script. Entonces necesitará un editor y nada mejor que 'vi'.

En este curso se le da mucha importancia a la cultura Unix. Para una persona que aprende 'vi' surgen varias oportunidades para aplicar lo que sabe en comandos distintos de 'vi'. La forma de trabajar de 'vi' ha tenido mucho impacto en otros programas. Puede encontrar juegos, hojas de cálculo utilidades, etc que usan cosas parecidas a 'vi'. Ya vimos 'sed' que usa comandos similares a algunos comandos de 'vi' y ahora vamos a ver a continuación algunas nociones de otro editor ('ed') que es otro buen ejemplo de lo que estamos comentando.

'ed' versus 'vi'
Dado que 'vi' está pensado para su uso interactivo desde un terminal no resulta muy apropiado de usar en combinación con otros programas mediante redirección de entrada salida. En el capítulo anterior de expresiones regulares vimos como 'sed' se podía usar como filtro de entrada salida. Existe un editor llamado 'ed' que es capaz de editar texto en modo no interactivo. Para ello se suele recurrir a un script que proporcionará a 'ed' las instrucciones adecuadas. Somos de la opinión de que las lecciones que no dejan ver rápidamente cual es su utilidad y que no se practican están condenadas al olvido. Por eso queremos evitar que piense en términos de *"para que necesito yo conocer esto de ed"*. Por eso más tarde comentaremos algo de 'diff' combinado con 'ed'. Así comprenderá mejor la utilidad del editor 'ed'.

Ahora vamos a generar un fichero 'ejemplo.txt' con unas pocas lineas para hacer unas pocas prácticas..

```
$ cd /tmp
$ cat <<FIN > ejemplo.txt
> No olvide visitar nuestra web
> http://www.ciberdroide.com
> y esto es todo.
> FIN
```

Vamos a sacar una copia para comprobar luego los cambios.

```
$ cp ejemplo.txt ejemplo.txt.original
```

Ahora generamos un fichero con las instrucciones para editar el fichero anterior. Deseamos que primero localice la linea que contiene 'www.ciberdroide.com' y a continuación añada una linea (comando 'a') '(Tienda Virtual de Linux)'. Salimos al modo comandos (comando '.') Salvamos (comando 'w') y salimos (comando 'q').

```
$ cat <<FIN > ejemplo.ed
> /www.ciberdroide.com/
> a
> (Tienda Virtual de Linux)
> .
> w
> q
> FIN
$ ed ejemplo.txt < ejemplo.ed

73
http://www.ciberdroide.com
99
```

Veamos como ha quedado.

```
$ cat ejemplo.txt

No olvide visitar nuestra web
http://www.ciberdroide.com
(Tienda Virtual de Linux)
y esto es todo.
```

Podemos ver las diferencias con el original usando el comando 'diff'. El comando diff es un comando que sirve para señalar diferencias entre dos ficheros pero con la opción -e estas diferencias se expresan en una forma adecuada para ser interpretadas por 'ed'. Veamos como funciona.

```
$ diff -e ejemplo.txt.original ejemplo.txt

2a
(Tienda Virtual de Linux)
.
```

Vamos a partir del fichero original y le vamos a pasar por la entrada estándar únicamente las diferencias obtenidas con 'diff -e'

```
$ cp ejemplo.txt.original ejemplo2.txt
$ diff -e ejemplo.txt.original ejemplo.txt > ejemplo.diff
$ cat ejemplo.diff

2a
(Tienda Virtual de Linux)
.
```

Vemos que falta las instrucciones 'w' y 'q' para salvar y salir respectivamente así que las añadimos al fichero 'ejemplo.diff'.

```
$ cat <<FIN >> ejemplo.diff
> w
> q
> FIN
$ cat ejemplo.diff

2a
(Tienda Virtual de Linux)
.
w
q
```

Ya tenemos todas las instrucciones necesarias. Vamos a aplicar los cambios con 'ed'.

```
$ ed ejemplo2.txt < ejemplo.diff

73
y esto es todo.
99

$ cat ejemplo2.txt

No olvide visitar nuestra web
http://www.ciberdroide.com
(Tienda Virtual de Linux)
y esto es todo.
```

En lugar de guardar varias versiones de un gran fichero que sufre pequeñas diferencias de vez en cuando se podría guardar una sola versión y unos pequeños ficheros con esas diferencias ocurridas entre una versión y la siguiente. De esta forma podrían aplicarse secuencialmente para ir recuperando sucesivamente las versiones anteriores en caso de necesidad. Cuando se desarrolla una aplicación se van introduciendo sucesivamente cambios en los programas y para poder recuperar versiones antigüas habría que ir salvando y conservando cada nueva versión. Los programas que se utilizan para el mantenimiento de distintas versiones de una aplicación suelen basarse en esta idea para ahorrar espacio de almacenamiento.

'vim' versus 'vi'

En Linux existe un editor 'vi' muy mejorado que se llama 'vim'. Este editor es muy similar a 'vi' pero con importantes mejoras. La más importante sin lugar a dudas es la capacidad de deshacer retrocediendo uno a uno en los últimos cambios realizados. Esto permite recuperar aquello de los destrozos que dijimos antes. También permite resaltar la sintaxis de ciertos ficheros como. shell-script, html, lenguaje C, y otros muchos lenguajes. Este y otros detalles lo hacen muy adecuado para programar. El manejo del cursor en los distintos modos del editor resulta mucho más cómodo y un montón de cosas más.

Mas adelante veremos que se pueden usar distintas opciones para que el editor funcione en la forma que nosotros queramos. 'vim' puede ejecutarse en modo compatible 'vi' (:set compatible) pero es mejor usar el modo no compatible que nos proporcionará las agradables ventajas sobre el 'vi' tradicional que antes comentamos. (:set nocompatible) Generalmente en Linux se instala 'vim' como sustituto de 'vi' de forma que aunque usted teclee 'vi' en realidad puede estar ejecutando 'vim' o 'elvis' u otros similares. Salgamos de dudas:

```
$ vi -? | less
```

Solo si le invita a usar la opción '-h' hágalo.

```
$ vi -h | less
```

Compruebe en las primeras lineas si se trata de 'vim'. Si ya ha entrado por error en editor salga mediante <ESC> :q<ENTER> Continuemos averiguando cosas.

```
$ whereis vi
$ whereis vim
$ whereis elvis
$ man vi
$ man vim
$ man elvis
$ man ex
$ man view
$ man ed
```

Ahora si vamos a entrar y a salir en el editor.

```
$ vi
```

Ya está dentro del editor. Compruebe si sale información relativa al tipo de editor y salga tecleando ':q' Para significar que estamos en el editor 'vi' pondremos el fondo azul oscuro y las letras en amarillo son las que salen en pantalla y en blanco las que introducimos nosotros.

```
~                       VIM - Vi IMproved
~
~                          version 5.3
~                    by Bram Moolenaar et al.
~
~                  Vim is freely distributable
~         type  :help uganda<Enter>     if you like Vim
~
~         type  :q<Enter>               to exit
~         type  :help<Enter>  or  <F1>  for on-line help
~         type  :help version5<Enter>   for version info
~
~
:q
```

Cuadro resumen para el uso de 'vi'

A continuación se describen una serie de comandos de uso frecuente en 'vi'. Se destacan los que se consideran más importantes con un mayor numero de asteriscos. En una primera lectura le recomendamos que empiece a utilizar y a aprender aquellos comandos marcados con '***' que son los más importantes. Le aconsejamos que no pase de lección hasta que no se aprenda como mínimo los comandos calificados con '***'. Al final del capítulo como es costumbre pondremos un test pero solo haremos preguntas sobre comandos calificados con '***' que serán los que practicaremos de forma especial en este capítulo.

Hemos agrupado toda esta información en un único cuadro para que sea su chuleta, para usar y aprender 'vi'. Solo se incluyen los comandos que suponemos serán de la mayor utilidad para un usuario novato y dado que hay varias formas de hacer una misma cosa hemos elegido por usted y hemos clasificado de forma totalmente subjetiva una serie de comandos para que inicialmente tenga que aprender solo un reducido número de ellos. También se incluyen otros comandos para que pueda ser usado como un cuadro de consulta bastante completo.

```
***       --> Comando de uso imprescindible.
**        --> Comando de uso muy frecuente.
*         --> Comando de uso no tan frecuente.
```

LINEA DE COMANDOS		
***	vi	Editar un nuevo fichero
***	vi <fichero>	Editar un fichero. (Si el fichero es de solo lectura lo editara en modo solo lectura.
*	vi <fichero1> <fichero2> <fichero3> ...	Editar varios ficheros sucesivamente.
**	vi -R <fichero>	Editar en modo solo lectura.
**	vi -r <fichero>	Editar recuperando modificaciones no salvadas.
MODO COMANDOS Se pasa desde el modo entrada (inserción o reemplazo) mediante <ESC>.		

Movimientos del cursor		
**	h	Carácter anterior
**	<ENTER>	Linea siguiente
**	j	Linea siguiente
**	k	Linea anterior
**	<ESPACIO>	Carácter siguiente
**	l	Carácter siguiente
*	L	Ultima linea de la pantalla
*	H	Primera linea de la pantalla
*	M	Linea central de la pantalla
*	<CTRL+B>	Retroceso de página
*	<CTRL+F>	Avance página
*	+	Primer carácter de la linea siguiente
*	-	Primer carácter de la linea anterior
*	$	Desplazarse al final de linea.
*	0	Desplazarse al principio de linea.
Modo entrada de texto		
***	i	Pasar a modo entrada insertando.
***	R	Pasar a modo entrada reemplazando.
**	a	Avanzar el cursor y pasar a modo entrada insertando.
***	o	Insertar una linea después de la actual y pasar a insertar.
*	O	Insertar una linea antes de la actual y pasar a insertar.
Corregir		
***	u	Deshacer última modificación. (En vim se pueden ir deshaciendo los los últimos cambios en otros editores vi solo se deshace el último cambio).
***	<CTRL+R>	En 'vim' sirve para deshacer el último comando 'u'
**	U	Recuperar linea completa.
Buscar		
**	/	Busca una cadena de caracteres, hacia delante.
*	?	Busca una cadena de caracteres, hacia atrás.
*	n	Siguiente en la búsqueda.
*	N	Anterior en la búsqueda.

Copiar, Mover, y borrar lineas		
***	dd	Borrar una linea.
***	<n>dd	borrar <n> lineas.
***	yy	Meter una linea en el buffer.
***	<n>yy	Meter <n> lineas en el buffer.
***	p	Copiar la linea del buffer después de la línea actual.
*	P	Copiar la linea del buffer antes de la línea actual.
Varios		
***	<CTRL+L>	Refrescar pantalla. (Útil cuando se descompone la pantalla)
**	<CTRL+G>	Visualiza el nombre del fichero, numero de lineas totales y en que linea estamos.
***	J	Une la línea actual con la siguiente.
**	.	Repetir último comando.
***	:	Pasar al modo ex (Modo comandos)
***	ZZ	Terminar salvando si hubo cambios.
MODO EX Se pasa desde el modo de comandos al modo ex con ':' Se puede forzar a entrar directamente en este modo usando la opción '-e' o ejecutando el comando 'ex'.		
**	:<#linea>	Ir a una linea.
*	:e <fichero>	Pasa a editar de forma simultanea otro fichero.
*	:e #	Volver al fichero anterior.
***	:w	Salvar. (No puede sobreescribir si se entró con vi -R)
***	:w!	Salvar forzando (Cuando falla :w).
**	:x	Salir salvando si es necesario.
***	:w <fichero>	Salvar con otro nombre. (No puede existir el fichero)
***	:w! <fichero	Salvar con otro nombre forzando.(Permite sobreescribir)
***	:r <fichero>	Lee un fichero y lo carga en el lugar indicado por la posición actual del cursor.
**	:r !<comando	Ejecuta en una subshell el comando y guarda el resultado del mismo en la posición indicada por el cursor.
***	:q	Salir si no hay modificación.
***	:q!	Salir sin salvar cambios.
**	:wq	Salir salvando cambios.

*	:sh	Ejecutar una subshell (se retorna con exit).
*	:g/<s1>/p	Visualizar lineas que tienen <s1>
*	:g/<s1>/s//<s2>/g	Sustituir globalmente <s1> por <s2>.
*	:g/<s1>/s//<s2>/gc	Ídem pero pidiendo confirmación.
*	:n	Siguiente documento.
*	:args	Mirar ficheros a editar.
*	:ab m n	Abreviaturas. Teclear 1 se sustituye por 2
*	:unab m	Desabreviar.
*	:map m n	Crear macro (m hace n) (Ver ejemplos con macros)
*	:map! m n	Crear macro en modo inserción (m hace n)
*	:unmap m	Destruir macro m
*	:unmap! m	Destruir macro de inserción m.
*	:cd	Cambiar de directorio actual.
**	:set nu	Activar visualización de números de línea
**	:set nonu	Activar visualización de números de línea
**	:set all	Ver todas las opciones con sus valores.
**	:set nocompatible	Opción de 'vim' para no obligar a ser compatible 'vi'
***	:set showmode	Para visualizar siempre el modo en que nos encontramos.
***	:!<comando>	Ejecutar un comando en una subshell.(Ver ejemplos con !)
Muchos de estos comandos pueden aplicarse dentro de un rango de líneas especificado.		Un número de línea (#linea) puede venir como:

Un punto	.	Línea actual
Un punto	.-N	N Líneas antes de la actual
Un punto	.+N	N Líneas después de la actual
Un número	N	Línea N
Un $	$	Ultima línea.
Un $	%	Desde la última a la primera línea (Abreviatura de 1,$).

***	:#linea1, #linea2 w <fichero>	Salvar desde 1 hasta la 2 en <fichero>
**	:#linea1, #linea2 co #linea3	Copiar desde 1 hasta 2 en 3.
**	:#linea1, #linea2 mo #linea3	Mover desde 1 hasta 2 en 3.
**	:#linea1, #linea2 de	Borrar desde 1 hasta 2.

**	:#linea1, #linea2 s/<s1>/<s2>	Sustituir la primera ocurrencia en cada linea desde 1 hasta la 2 la cadena <s1> por la cadena <s2>
**	:#linea1, #linea2 s/<s1>/<s2>/c	Sustituir la primera ocurrencia en cada linea desde 1 hasta la 2 la cadena <s1> por la cadena <s2> con confirmación.
**	:#linea1, #linea2 s/<s1>/<s2>/g	Sustituir todas las ocurrencias en cada linea desde 1 hasta la 2 la cadena <s1> por la cadena <s2>
**	:#linea1, #linea2 s/<s1>/<s2>/gc	Sustituir todas las ocurrencias en cada linea desde 1 hasta la 2 la cadena <s1> por la cadena <s2> con confirmación

Sería conveniente que sacara por la impresora esta información para usarla a modo de chuleta en sus primeras prácticas con 'vi'.

Existe otra forma de aplicar comandos dentro de un rango de lineas. Desde el modo de comandos pulsando 'v' se pasa al modo 'vi' o modo 'visual'. Entonces moviendo el cursor marcamos todo el rango de lineas que queramos tratar y pulsando ':' pasamos al modo 'ex' para aplicar el comando que queramos. Cualquier comando que se pueda introducir en 'vi' estando en el modo de 'ex' puede ser incluido en un fichero .exrc en el $HOME que servirá para que 'vi' ejecute esos comandos cada vez que arranque. Se usa mucho para definir macros y para activar determinadas opciones con set pero eso lo veremos en otro capítulo.

Practicando lo más esencial de 'vi'.
Si no se usa ninguna opción inicial vi empezará en modo comandos. Para pasar al modo entrada hay varias formas pero recomendamos que se aprenda de momento solo. i, a, o, R. Ya hemos descrito estas funciones en 'chuleta-vi.txt' y practicaremos con ellas más adelante.

Del modo entrada (Insertar o reemplazar) se sale con <ESC>.

Normalmente trabajábamos en '/tmp' como directorio temporal pero esta vez vamos a crear un directorio nuestro de trabajo para poder conservar nuestro trabajo el tiempo que queramos. Para ello crearemos un directorio 'cursolinux' y nos situaremos en él para trabajar.

```
$ cd
$ mkdir cursolinux
$ cd cursolinux
$ vi ej-vi1.txt
```

Dado que no existía ningún fichero 'ej-vi1.txt' estaremos editando un fichero nuevo y dependiendo del editor usado (nosotros usaremos 'vim') aparecerá algo del tipo.

```
~
~
~
"ej-vi1.txt" [New File]
```

Está en modo comandos y va a teclear sus dos primeros comandos. Cuando vea '<ENTER>' en los ejemplos significa que debe pulsar esa tecla.

```
:set showmode<ENTER>
:set nu<ENTER>
```

Acabamos de activar la visualización del modo de funcionamiento y la visualización de los números de línea. Ninguna de estas cosas son necesarias para editar un fichero pero a nosotros nos van a ayudar a ver más claro lo que hacemos.

Ahora pulsaremos la 'i' y aparecerá en la linea inferior el indicador de que estamos en modo entrada para inserción. Desde este modo introduciremos las siguientes lineas de texto.

```
  1  Estamos practicando con el editor vi.<ENTER>
  2  No parece tan difícil de usar.<ENTER>
  3  Ya voy por la tercera linea y no ha ocurrido nada grave.<ENTER>
  4  <ENTER>
  5  <ENTER>
  6  Como he saltado un par de líneas voy por la linea seis.<ENTER>
  7  De momento no escribo nada más.<ESC>
ZZ
```

Con la tecla <ESC> salimos al modo comandos y con 'ZZ' (mayúsculas) desde el modo comandos salimos salvando el fichero. Ahora vamos a volver a editarlo.

```
$ vi ej-vi1.txt
```

Volvemos a activar los modos 'showmode' y 'nu' y nos situamos en la línea 3 y la borramos con 'dd' luego nos situamos en la primera línea y hacemos 'p' que copiará la linea eliminada a continuación de la línea primera. 'dd' guarda siempre en un buffer la información eliminada para poder recuperarla con 'p'. Podríamos haber borrado cinco lineas de golpe usando '5dd' y las cinco líneas habrían sido guardadas en el buffer para su posible recuperación con 'p'.

```
:set showmode<ENTER>
:set nu<ENTER>
:3<ENTER>
dd
:1<ENTER>
p
```

Pero ahora la línea 2 dice que es la tercera así que intente usted mismo borrar una de las líneas vacías y situarla como segunda línea usando de nuevo 'dd' y 'p'. Esta vez mueva de linea usando 'j' desde el modo comandos para bajar a la linea siguiente y 'k' para subir a la línea anterior. Si el teclado está configurado en su editor para manejar correctamente las teclas de flechas podrá hacer esto mismo usando las teclas flecha abajo y flecha arriba en sustitución de la 'j' y la 'k' respectivamente. Si algo sale mal borre el fichero y vuelva a empezar procurando no volver a meter la pata. Bueno el caso es que el fichero debería quedar con este aspecto.

```
  1 Estamos practicando con el editor vi.
  2
  3 Ya voy por la tercera linea y no ha ocurrido nada grave.
  4 No parece tan difícil de usar.
  5
  6 Como he saltado un par de líneas voy por la linea seis.
  7 De momento no escribo nada más.
```

Vamos a sacar una copia de este fichero modificado con otro nombre, y luego vamos a duplicar varias veces el contenido de las líneas cuatro y cinco usando '2yy' (guarda en el buffer dos lineas) y 'p' para (copia el contenido del buffer en la línea actual). Asegurese que está en modo comandos. Pulse <ESC> si no está seguro y teclee los comandos siguientes. Advierta que algunos comandos empiezan por ':' y terminan por <ENTER> mientras que otros no.

```
:w ej-vi2.txt
:4<ENTER>
2yy
:3<ENTER>
p
:3<ENTER>
p
:3<ENTER>
p
```

Estamos repitiendo varias veces la línea 'No parece tan difícil de usar.' Debería quedar finalmente con este aspecto.

```
  1 Estamos practicando con el editor vi.
  2
  3 Ya voy por la tercera linea y no ha ocurrido nada grave.
  4 No parece tan difícil de usar.
  5
  6 No parece tan difícil de usar.
  7
  8 No parece tan difícil de usar.
  9
 10 No parece tan difícil de usar.
 11
 12 Como he saltado un par de líneas voy por la linea seis.
 13 De momento no escribo nada más.
```

Como hemos visto con '2yy' hemos metido en el buffer dos lineas en lugar de una. Para meter una sola linea habría bastado usar 'yy' o '1yy'. Ahora vamos a salir pero sin guardar ninguno de los cambios realizados. Primero intentaríamos salir sin más con ':q' desde el modo comandos.

```
:q
```

Obtendremos un aviso del editor para no perder las últimas modificaciones.

```
No write since last change (use ! to override)
```

Pero realmente deseamos salir sin salvar así que intentamos de nuevo con ':q!' desde el modo comandos.

```
:q!
```

Vamos a comprobar que los últimos cambios no se han guardado, y vamos a ver que permisos tiene este fichero.

```
$ cat ej-vi1.txt

Estamos practicando con el editor vi.
No parece tan difícil de usar.
Ya voy por la tercera linea y no ha ocurrido nada grave.


Como he saltado un par de líneas voy por la linea seis.
De momento no escribo nada más.
$ ls -l ej-vi1.txt

-rw-r--r--   1  xxxx xxxx   216 ago 12 18:24 ej-vi1.txt
```

La información de permisos puede ser diferente. Recuerde que dijimos que los permisos de un fichero cuando se crea dependen de la 'umask' pero ahora solo nos interesa comprobar efectivamente tenemos permisos para leer y escribir en este fichero. Vamos a eliminar los permisos de escritura del fichero y vamos a intentar editarlo nuevamente.

```
$ chmod 444 ej-vi1.txt
$ ls -l ej-vi1.txt
-r--r--r--   1  xxxx xxxx   216 ago 12 18:24 ej-vi1.txt
$ vi ej-vi1.txt
```

Probablemente aparecerá en la linea inferior de su editor una advertencia de que está en modo solo lectura.

```
"ej-vi1.txt" [readonly] 7 lines, 216 characters
```

Bueno a pesar de ello vamos a intentar hacer cambios y como estamos practicando continuaremos con una serie de comandos que aún no hemos visto y que teníamos clasificados con '***'.

```
:3
:.,$ w trozofinal
:1,. w trozoinicial
:1,$ de
:r trozoinicial
:$
```

```
:r !date
:$
:r trozofinal
:w tempfich
:w

'readonly' option is set (use ! to override)
```

Permitió escribir otros ficheros pero se niega a sobrescribir el fichero que estamos editando con los cambios porque está en modo solo lectura y sugiere que usemos '!' para forzar su escritura.

```
:w!
```

En este caso estamos forzando la situación y estamos escribiendo en un fichero en el que no tenemos permisos pero puesto que somos propietarios del mismo el editor puede manejar los permisos para dar permiso de escritura, sobrescribir el fichero y volver a cambiar los permisos dejando los permisos originales.

Podemos ejecutar comandos desde el editor así que porque no miramos los permisos con 'ls -l'.

```
:!ls -l

total 6
-r--r--r--   1 xxxx xxxx       304 ago 12 19:11 ej-vi1.txt
-r--r--r--   1 xxxx xxxx       304 ago 12 19:11 ej-vi1.txt~
-rw-r--r--   1 xxxx xxxx       216 ago 12 18:28 ej-vi2.txt
-rw-r--r--   1 xxxx xxxx       304 ago 12 19:09 tempfich
-rw-r--r--   1 xxxx xxxx       147 ago 12 19:06 trozofinal
-rw-r--r--   1 xxxx xxxx       126 ago 12 19:06 trozoinicial

Press RETURN or enter command to continue
```

Debemos pulsar <ENTER> para volver al editor Puesto que podemos ejecutar cualquier comando también podemos ejecutar bash como una subshell. Aprovecharemos para ver los procesos activos.

```
:!bash
bash-2.01$ ps

  PID TTY STAT TIME COMMAND
  221   2 S    0:00 -bash
  893   2 S    0:00 vi ej-vi1.txt
  944   2 S    0:00 bash
  951   2 R    0:00 ps
bash-2.01$ exit

exit

1 returned

Press RETURN or enter command to continue
```

Por fin estamos de vuelta en el editor. Vamos a continuar practicando algunos comandos importantes que nos quedan.

```
:3<ENTER>
o
Hemos pasado a insertar abriendo una linea.<ESC>
J
```

Con la 'J' mayúscula hemos unido dos líneas en una. Ahora vamos a pasar al modo entrada para reemplazar.

```
:3<ENTER>
R
XXXXXXXXXXXXXXXXX<ESC>sertar abriendo una linea.  Ya voy por la tercera linea y no ha
ocurrido nada grave.
```

```
<ESC>
<CTRL+G>


"ej-vi1.txt" [Modified][readonly] line 4 of 10 --40%-- col 44
```

Desde el modo comandos podemos consultar el nombre del fichero, cuantas lineas tiene en total, en que línea y columna estamos, y otras cosas con solo pulsar <CTRL+G> .

Ahora utilice varias veces el comando 'u'. Si está en 'vim' verá como se deshacen retrocediendo secuencialmente en las últimas modificaciones que hemos realizado. Luego si está usando 'vim' use <CTRL+R> para avanzar de nuevo en las últimas modificaciones hasta recuperarlas todas. Para terminar salga con ':q!'

Practique con la combinación de 'yy' o 'nyy' y 'p'. O con la combinación 'dd' o 'ndd' y 'p'. Se usan mucho para copiar o trasladar pequeños bloques de información.

Para mover una linea hacer 'dd', luego situarse en la posicion destino y hacer 'p'. Para mover varias lineas hacemos algo parecido. Borramos primero varias lineas y luego desde la posición destino hacemos 'p'. Para copiar en lugar de mover usaríamos 'yy' en lugar de 'dd'.

Bueno utilice su propia imaginación e investigue y practique por su cuenta. Esta lección no le obliga a digerir conceptos nuevos pero si requiere iniciativa para practicar con el editor.

Existe un comando que lo hemos marcado como esencial cuando en realidad se usa poquísimo pero es un comando que conviene conocer porque en el momento más inesperado puede ser necesario usarlo. Con este comando el editor redibuja totalmente el contenido de la pantalla. Esto puede ser necesario porque a su terminal podría llegarle información desde otro proceso. Por ejemplo un mensaje de error del sistema o un aviso de su administrador para que finalice antes de 10 minutos. Quizás su administrador es de los que repite el mensaje más de una vez y a cada mensaje recibido su pantalla queda totalmente destrozada. No pasa nada. Pulse <CTRL+L> y todo volverá a la normalidad. Esto puede ocurrirle cuando este usando otro programa distinto de 'vi' y quizás pueda usar gualmente <CTRL+L>.

Recuperar la información perdida por muerte prematura del editor
Otra cosa que debe saber es que cuando 'vi' recibe una señal para finalizar ordenadamente guardará el contenido de lo que está editando en un fichero de seguridad. Para recuperar esta información necesitará usar 'vi -r fichero'. Como ejercicio libre le proponemos que empiece a editar un fichero que llamaremos 'prueba-kill.txt'. Escriba unas cuantas cosas y sin salvar nada, pase a otro terminal y envíe un kill con el valor por defecto (15 = SIGTERM). Para eso primero tendrá que averiguar el pid del proceso vi. Por ejemplo use 'ps | grep vi'. Supongamos que averigua que su pid es 445. Bastaría hacer 'kill 445'. Esto finalizará la edicion del fichero 'prueba-kill.txt'. No edite este fichero con 'vi prueba-kill.txt' porque perderá todos los cambios. En cambio si edita con 'vi -r prueba-kill.txt' si los recuperará. En este punto salve y continue editando cosas y antes de salvar los nuevos cambios vuelva a averiguar el pid desde otro terminal. Supongamos que el nuevo pid fuera 448. Envíe esta vez la señal SIGKILL no enmascarable mediante 'kill -9 448'. Comprobará que esta vez el editor no será capaz de recuperar los cambios. Por ejemplo si ocurre un apagado normal (shutdown) del sistema mientras está usted editando o mientras usted está ausente del terminal con la sesión de edición abierta, tampoco ocurrirá nada. La razón de esto, es que el sistema primero envía a todos los procesos activos una señal de SIGTERM para que finalicen ordenadamente. Después de esto y pasados unos segundos se supone que la mayoría de los procesos habrán terminado. (A los restantes procesos les enviará un SIGKILL para matarlos incondicionalmente). Cuando le ocurra esto tendrá que volver a entrar en sesión con login y seguramente recibirá una notificación en su correo advirtiendo de la existencia de un fichero que debe ser editado con la opción '-r' para recuperar información volcada durante la finalización prematura de 'vi'.

EL EDITOR VI (segunda parte)

- Objetivos de esta lección
- Búsquedas
- Sustituciones
- Edición de varios ficheros secuencialmente
- Edición simultanea de ficheros
- Visualización de caracteres de control
- Introduciendo caracteres de control en el texto
- El comando tilde
- Operaciones con bloques.
- Repetición del último comando
- Tabulación
- Abreviaturas
- Macros
- Opciones de vi
- Configuración del editor
- Tag
- Test

Objetivos de esta lección
Esta lección menciona algunas posibilidades avanzadas del editor 'vi'. Se supone que el alumno ha estudiado las lecciones precedentes y en particular el capítulo anterior dedicado al editor 'vi'. Eso significa que ya sabe editar. Vamos a suponer que incluso ha practicado un poco por su cuenta. Por ello lo que vamos a explicar en este capítulo será de escasa utilidad si no ha asimilado la lección anterior.

Poco a poco vamos a ir asumiendo un mayor dominio del alumno. Ya no vamos a detallar tan a menudo paso por paso cosas que ya se han explicado y practicado más de una vez. Eso le obligará a poner un poco más de iniciativa por su parte.

Los ejemplos de las lecciones anteriores eran prácticas totalmente guiadas, comando a comando, pero a partir de ahora tendrá que probar cosas no necesariamente idénticas a los ejemplos que nosotros utilicemos.

No se preocupe. Seguro que lo hará muy bien. Si ha llegado hasta aquí es señal de haber conseguido un avance significativo en sus conocimientos, y como Linuxero se encuentra quizás a medio camino entre el novato bruto y extremadamente torpe y el repugnante listillo informático. :-)

Confiamos plenamente en su capacidad pero recuerde que nosotros no somos responsables de lo que pueda pasarle a su ordenador. Continúe practicando con un usuario distinto de root y un directorio que no contenga información valiosa.

Seguramente en esta lección aprenda cosas que algunas personas que llevan usando 'vi' durante varios años aún no conocen, pero también contiene cosas que pueden ser muy importantes para usted. No todos usamos las mismas cosas. La mayoría de los documentos que explican el uso de 'vi' lo hacen de forma muy incompleta. Nosotros no vamos a hacer un tratado completo de este editor porque necesitaríamos muchas lecciones y nos saldríamos de los objetivos del curso pero si procuraremos mostrarle un amplio abanico de posibilidades para que tenga donde elegir. En la primera lección sobre 'vi' le mostramos un juego reducido de comandos pero suficiente para defenderse. En esta lección le damos la posibilidad de convertirse en un virtuoso de este editor.

Búsquedas

Desde el modo comandos se puede usar '/' para buscar. Admite expresiones regulares. Por ejemplo si introduce '/debian.com' podría localizar una cadena 'debian-com' o 'debian:com' y no solo 'debian.com'. Recuerde que algunos caracteres como el punto el asterisco, etc tienen un significado especial y para usarlas en 'vi' hay que precederlas de '\' que actuará como carácter de escape. También hay que escapar los caracteres como por ejemplo '/' y '&' porque sin tener significado como operador de expresión regular si tiene un significado especial para 'vi' que veremos cuando expliquemos las sustituciones. No vamos a volver a explicar las expresiones regulares porque ya dedicamos una lección. Practique lo que va aprendiendo y recuerde que las expresiones regulares no funcionan igual en todas las aplicaciones. Vamos a poner varios ejemplos para su uso en búsquedas desde 'vi'. Para buscar '</TABLE>' habría que usar '/<\/TABLE>'.

Es decir que algunos caracteres no los podemos usar tal cual para buscarlos sino que hay que es caparlos. En el caso anterior hemos escapado '/'. Tenga en cuenta esto cuando intente por ejemplo buscar un carácter '$' en un texto.

Con 'n' buscará la ocurrencia siguiente de la cadena en el fichero. Con 'N' buscará la ocurrencia anterior. Una vez llegado al final comienza de nuevo la búsqueda por el principio.

También se puede empezar la búsqueda desde el primer momento en sentido ascendente usando '?' en lugar de '/'. En este caso los comandos 'n' y 'N' también funcionaran en sentido contrario que con el comando '/'.

Si usamos '/' o '?' sin nada más a continuación, se repetirá la última búsqueda.

Las expresiones regulares ofrecen grandes posibilidades como ya hemos visto pero particularmente interesantes para 'vi' pueden ser las búsquedas a principio o a final de una línea. En el ejemplo buscaremo 'El' a principio de una línea y luego 'as' al fineal de línea.

```
'/^El'
'/as$'
```

Sustituciones
Ya dispone de un resumen de todos los comandos de 'vi' que proporcionamos en el capítulo anterior así que lo mejor es presentar una serie de ejemplos explicando para que sirven.

Vamos a poner tres formas distintas de sustituir 'encarnado' por 'colorado' en las tres primeras líneas de un texto, y solo deseamos que ocurra la sustitución en la primera ocurrencia de cada línea caso. Es decir si una línea contiene varias veces la palabra 'encarnado' solo deseamos sustituir la primera.

```
:1,3 s/encarnado/colorado/
:1,3 s:encarnado:colorado:
:1,3 sçencarnadoçcoloradoç
```

Ahora de dos formas distintas vamos a sustituir en todo el texto 'encarnado' por 'colorado'. Es decir si una linea contiene varias veces la palabra 'encarnado' deseamos que la sutitución se realice en todas ellas. Para eso utilizaremos un comando similar al anterior pero finalizado en '/g'. Los ejememplos solo varían en la forma de indicar con '1,$' o con '%' la búsqueda en todo el texto.

```
:1,$ s/encarnado/colorado/g
:% s/encarnado/colorado/g
```

Ahora si sutituimos en todo el texto 'encarnado' por 'colorado' pero confirmando cada cambio manualmente. Para eso utilizaremos un comando similar al anterior pero finalizado en '/gc'. La 'g' tiene el mismo significado que en el caso anterior y la 'c' significa 'confirmación'.

```
:1,$ s/encarnado/colorado/gc
```

Vamos a usar ahora las sustituciones con expresiones regulares.

1) Poner a doble línea todo un texto.

```
:%s/$/\
/g
```

2) Eliminar blancos y tabuladores a final de linea.

```
:%s/[space tab]*$//
```

3) Eliminar blancos y tabuladores al principio de línea.

```
:%s/^[space tab]*//
```

4) Sustituir cualquier secuencia de blancos y tabuladores por un único blanco.

```
:%s/[space tab][space tab]*/ /
```

5) Poner entre paréntesis todos los números enteros de un texto.

```
:%s/[0-9][0-9]*/\(&\)/g
```

En este último hemos usado el operador '&' en la misma forma que vimos con 'sed' unas lecciones atrás. Se pueden hacer en poco tiempo cosas bastante complejas. Por ejemplo vamos a hacer un shell-script que genere otro shell-script. Para ello primero lo que haremos es escribir el programa que deseamos generar.

```
echo 'kkkk'
echo 'kkkk'\''hhhh'
```

Para generarlo bastará tratar cada línea poniendo un "echo '" al comienzo de cada línea y una comilla simple al final "'" de cada línea, pero antes escaparemos cualquier posible comilla simple que pudiera existir. Para ello editamos el texto anterior y hacemos lo siguiente:

```
:% s/\'/\'\\\'\'/g
:% s/^/echo \'/
:% s/$/\'/
```

Deberá quedarnos de esta forma.

```
echo 'echo '\''kkkk'\'''
echo 'echo '\''kkkk'\''\'\'''\''hhhh'\'''
```

Bueno lo hemos realizado con 'vi' y a pesar de lo complicado del resultado solo hemos realizado tres operaciones sencillas. Ahora ejecute este fichero y verá como obtiene el primero.

Bueno si no conoce C no le entusiasmará este otro ejemplo pero eso de hacer programas que generan programas nos sirve para practicar las sustituciones usando 'vi'. Podemos escribir el programa en C que deseamos generar y luego tratarlo sustituyendo todas las '\' por '\\' y todas las '"' por '\"'. Es decir escapamos esos dos caracteres que darían problemas más tarde. Luego añadimos a principio de cada línea 'printf('" y finalizamos cada línea con '\n");'

Las sustituciones que acabamos de mencionar serían de esta forma.

```
:% s/\\/\\\\/g
:% s/"/\\"/g
:% s/^/printf("/
:% s/$/\\n");/
```

Edición de varios ficheros secuencialmente
Al editor 'vi' se le puede pasar como argumento no solo un fichero sino varios para ser editados en secuencia. Por ejemplo:

```
$ vi fich1 fich2 fich2
```

Bueno esto puede ser más útil de lo que parece a simple vista. Vamos a suponer que tenemos que modificar una gran cantidad de ficheros para sustituir en muchos de ellos la cadena 'acastro' por la cadena 'A.Castro' pero tampoco lo podemos hacer de forma automática porque nos conviene ver caso a caso. Por eso pasamos a 'vi' el resultado de una búsqueda con 'grep'.

```
$ vi `grep -il acastro *.txt`
```

Una vez dentro del editor hacemos lo siguiente.

```
:args
```

El comando ':args' nos informará del nombre de los ficheros que vamos a editar secuencialmente. Intente probar esto creando sus propios ficheros de prueba. Ya sabe como hacerlo. Conviene que todos ellos tengan una cadena idéntica para ser localizada con grep. Una vez que el editor 'vi' este preparado para editar secuencialmente varios ficheros busque la cadena con el operador '/' y aunque pasemos a editar el siguiente fichero se conservará almacenada la cadena de búsqueda. Esta forma de trabajar facilita mucho la modificación de varios ficheros.

Para editar ficheros secuencialmente le conviene tener en cuenta los siguientes comandos. Algunos no los habíamos comentado todavía.

```
:args   Obtiene la lista de los ficheros
:next   Pasa al fichero siguiente
:prev   Vuelve al fichero anterior
:rewind Vuelve al primero
:rewind! Vuelve al primero abandonando los cambios en el fichero actual
:w      Salva los cambios del fichero anterior
:q      Sale del editor. (Solo si ya se han editados todos)
:q!     Sale del editor incondicionalmente.
```

Edición simultanea de ficheros

Podemos pasar de la edición de un fichero a la edición de otro usando ':e fichero' pero antes deberemos salvar el contenido de la edición actual. También podemos forzar el paso a editar otro fichero con ':e! fichero' pero en ese caso se perderán las modificaciones realizadas. Usando ':e#' volvemos a retomar la edición del fichero anterior situados en la misma línea donde lo dejamos. Intente practicar lo siguiente editando unos ficheritos de prueba.

```
:e fich1
:w
:e fich2
:e! fich3
:e#
```

Visualización de caracteres de control

En 'vi' los caracteres de control se visualizan como un acento circunflejo y una letra. Por ejemplo <CTRL+K> se visualizará como '^K' y la manera de distinguir un carácter circunflejo real seguido de una 'K' es moviendo

el cursor para ver si de desplaza uno o dos caracteres cuando pasamos por el carácter circunflejo. También se usa con similar significado el carácter tilde '~' pero para caracteres de 8 bits.

Introduciendo caracteres de control en el texto
Imaginemos que deseamos introducir un carácter de control para provocar un salto de página en la impresora. Para ello deberemos introducir un <CTRL+L> que será visualizado como '^L'. La forma de hacerlo es pulsar desde el modo inserción el carácter <CTRL+V> y a continuación el carácter <CTRL+L>. Es decir el carácter <CTRL+V> actúa como carácter de escape. Podemos pulsar <CTRL+V> y luego una tecla de función, u otras teclas especiales.

También se pueden buscar caracteres de control. Para ello tambíen hay que introducirlos usando <CTRL+V>.

Edite un fichero de prueba y proceda a poner varios saltos de página en distintos lugares. Luego busque con '/' y con '?' los saltos de página. Pruebe a insertar los códigos de las teclas <ENTER> y <ESC>, mediante <CTRL+V><ENTER> y <CTRL+V><ESC> respectivamente. Deberán aparecer como '^M' y como '^['.

El comando tilde
El comando '~' sustituye el carácter situado bajo el cursor de mayúscula a minúscula y viceversa avanzando luego el cursor.

Operaciones con bloques
Se pueden marcar hasta 26 posiciones ('a'..'z') para un conjunto de uno o mas ficheros editados simultáneamente. Permite trabajar marcando principio y fin de bloques para mover, copiar o borrar en varios ficheros. Se pueden usar las marcas en cualquier comando en sustitución de '.', '$', o los números de línea como ya hemos visto. La ventaja es que si estamos insertando o borrando líneas tendríamos que volver a comprobar los números de líneas. Por el contrario si hemos colocado una marca en una línea continuará en esa línea mientras no la eliminemos. Para colocar una marca podemos situarnos con el cursor en la línea deseada y luego hacer ':k <letra-minúscula>' ':ma <letra-minúscula>' ':ma <letra-minúscula>' o directamente sin pasar al modo ex mediante 'm <letra-minúscula>'. Para practicar con ello edite un fichero de prueba y coloque varias marcas en distintas líneas. Luego realice distintas operaciones como por ejemplo el borrado de bloques. Compruebe como desaparece una marca cuando se borra la línea que la contiene. Las marcas no se ven pero si intenta usar una marca que no existe obtendrá un error. Compruebe como reaparece la marca cuando deshace la operación con 'u' (undo). Algunas formas de utilización de marcas son las siguientes.

```
:'a,'b de
:'b,'b+1 de
:'a,. de
:'a,'b co 'c
:'a,'b mo 'c
```

También podemos posicionarnos en el lugar de una marca pulsando comilla-simple y <letra-minúscula> desde el modo comandos. Un detalle interesante es que siempre podemos volver a la línea donde nos encontrábamos, la última vez que dimos un salto de posición en el texto. Para ello pulsamos '' (comilla-simple dos veces).

Repetición del último comando
Cuando hay que hacer varias veces la misma cosa resulta tremendamente útil poder repetir el último comando. Imaginemos que desde el modo de comandos hacemos en una línea 'RLínea <ESC>' para sustituir 6 caracteres por la cadena 'Línea ' Inmediatamente después de ello y antes de hacer ninguna otra cosa podríamos situarnos en otra línea y pulsar '.' (punto) para repetir ese mismo comando y luego cambiar a otra y otra línea y volver a

pulsar '.' punto a cada vez para efectuar el mismo cambio en varios sitios. Tambíen mencionamos la posibilidad de ejecutar un comando del sistema operativo lanzando una sub-shell. Pues bien este tipo de comandos ': <comando>' puede ser repetido usando ':!!'

Tabulación

El carácter con la tecla <TAB> o con el carácter <CTRL+I> insertará un carácter de tabulación. El ancho de la tabulación será por defecto igual a 8 espacios. Esto puede ser variado por ejemplo a 5 con ':set tabstop=5'. Si deseamos añadir un tabulador al principio de línea en las siguientes 7 lineas a la posición del cursor podemos hacer. '7>>'. Tambíen se podría haber hecho lo mismo con ':.,.+7 >>'. Para insertar un tabulador a principio de cada línea del texto sería ':% >>' Cuando se usa el editor para editar fuentes de programas puede resultar útil usar la indentación automática. 'set ai' para eliminar la indentación automática usaremos 'set noai'. Si en alguna ocasión necesita sustituir todos los tabuladores por caracteres en blanco use el comando 'expand'. No es un comando de 'vi' sino del SO. Consulte con el man su sintaxis y las distintas opciones. Para hacer una prueba edite un fichero insertando tabuladores y luego conviertalo de la siguiente forma.

```
expand fichero > ficheroexpandido.
```

Abreviaturas

En 'vi' hay muchos detalles pensdos a ahorrar pulsaciones de declas. Vamos a ver el comando ':ab' Si está acostumbrado al uso de abreviaturas le resultará muy práctico. Puede definir las abreviaturas que quiera y una vez tecleadas en modo entrada de dados (insercion o sobrescritura) serán sustituidas por la cadena que desee. Vamos a usar dos abreviaturas una 'ci' para 'Ciberdroide Informática' y otra 'INMHO' para 'en mi humilde opinión'. También vamos a listar las abreviaturas definidas.

```
:ab ci Ciberdroide Informática
:ab INMHO en mi humilde opinión
:ab
```

Use sus propias abreviaturas en un texto y observe como la abreviatura no será interpretada si la secuencia de caracteres forma parte de una palabra.

Macros

Es posible programar macros. Por ejemplo vamos a hacer una macro que mueva 4 líneas al final del fichero

```
:map xxx 4dd:$^Mp
:map
```

Con esta macro ocurrirá algo. En primer lugar si teclea las tres 'x' seguidas se ejecutarán los comando '4dd' (borrar cuatro lineas), ':$<ENTER>' (ir a la última línea) y 'p' (recuperar el contenido del buffer) Si pulsamos una sola 'x' notaremos que no pasa nada inmediatamente pero luego se eliminará un carácter tal y como ocurre normalmente. Es decir la macro tiene un tiempo para ser tecleada completamente en caso contrario no actúa. Para ver todas las macros definidas usaremos ':map'

Podemos eliminar la macro que acabamos de crear con 'unmap'

```
:unmap xxx
```

Muchas veces se usan las macros para definir comandos asociados a teclas especiales. Para ello hay que conocer la secuencia de caracteres que envían esas teclas. Si en modo inserción usamos <CTRL+V><F2> obtendremos '^[[[B'. Pero también podemos usar '<F2>' para definir macros. Vamos a poner algunos ejemplos prácticos.

F2 (Filtro PGP) Firmar.

```
:map □[[B 1G!Gpgp -satf 2>/dev/tty
```

Sería lo mismo si usaramos '<F2>' en lugar de '^[[[B'.

```
:map <F2> 1G!Gpgp -satf 2>/dev/tty
```

F3 (Filtro PGP) Encriptar

```
:map □[[C 1G!Gpgp -eatf 2>/dev/tty
```

F4 (Filtro PGP) Desencriptar y comprobar firma

```
:map □[[D 1G!Gpgp  2>/dev/tty; sleep 7
```

F5 (Filtro ispell) Corrector ortográfico.

```
:map □[[E :w
:!ispell %
:e!
```

Opciones de vi

Sería absurdo pretender explicar en una lección todo el editor 'vi'. Tampoco hay necesidad de ello. Hay montones de opciones y ya hemos visto algunas. Si el editor es 'vim' en lugar de 'vi' el número de posibilidades es aun mayor. Puede ver las opciones disponibles y sus valores actuales usando el comando ':set all' desde el modo de comandos. No se asuste. No ha que conocerlas todas. Hay opciones que seguramente la persona que las implementó ya ha olvidado para que sirven. Una opción se puede indicar muchas veces de dos formas. Una en forma completa y otra en forma abreviada. Por ejemplo ':set nu' en lugar de ':set number' y para desactivar la opción se suele usar la misma opción precedida de 'no'. Por ejemplo ':set nonu' o ':set nonumber'. Las opciones que deben tomar un valor numérico se establecen con el nombre de la opción seguida de '=' y el valor. Por ejemplo ':set tabstop=5'

Configuración del editor

El fichero más estándar para configurar 'vi' en cualquier SO de tipo Unix es '~/.exrc'. Como alternativa se puede usar ficheros más específicos para el editor concreto que estemos usando. Por ejemplo en el caso de 'vim' tenemos '~/.vimrc'. Compruebe si existe un ejemplo '/etc/vimrc' en su Linux y mire su contenido. Se trata de un fichero de ejemplo para copiarlo en ' ~/.vimrc' Contiene algunas configuraciones que pueden resultar útiles. El carácter comilla doble '"' se usa para considerar el resto de la línea como un comentario. Si considera

que el editor tiene un comportamiento que usted considera incómodo quizás se deba a la existencia de algunas opciones que no se adaptan al uso que usted da a su editor. Por ejemplo para editar programas C puede resultar cómodo la auto- indentación pero no le gustará que parta las líneas automáticamente cuando excedan un determinado ancho. También es buena idea incluir en su fichero de configuración las abreviaturas y las macros que desee usar. Ya hemos visto algunas y seguramente ya se ha preguntado como hacer para no tener que introducirlas cada vez. Es bueno perder un poco de tiempo en personalizar el editor.

Tag

Para editar fuentes C por ejemplo resulta muy practico. Supongamos que tenemos un fuente llamado pingpong.c. Crearemos en primer lugar un fichero tags ejecutando "ctags pingpong.c" (ver ctags(C)). Se crea un fichero tags que contendrá por ejemplo lo siguiente:

```
plot    pingpong.c      ?^plot(x, y, ch)$?
salir   pingpong.c      ?^salir(s)$?
```

Con ello se referencian las funciones. Para editar el fichero podemos hacer vi -tplot. Aparecera el cursor en la posición de esa función. Si introducimos :ta salir desde el modo de comandos nos situaremos en esa función. Se pueden crear tags para varios ficheros simultáneamente mediante ctags */*.[cfg] (ver ctags(C)). De esta forma podríamos editar cómodamente varios fuentes.