Curso de GNU/Linux para principiantes

Margarita Manterola Maximiliano Curia

Facultad de Ingeniería - UBA
Actualizado Octubre 2004

1. Conociendo GNU/Linux

GNU/Linux es un sistema operativo derivado de UNIX, que se distribuye en forma libre.

1.1. Qué es UNIX

UNIX es un sistema operativo multitarea, multiusuario, creado en 1969 por los investigadores Thompson y Ritchie de los Laboratorios Bell, en los Estados Unidos. Las primeras versiones fueron escritas en assembler, pero muy pronto fue re-escrito en lenguaje C.

En sus primeros años, no se lo utilizó comercialmente, sino que se lo usaba para proyectos de investigación en laboratorios y se distribuía gratuitamente en las universidades, donde tuvo mucha aceptación.

En 1975, Bell decidió comercializarlo. Dado que el sistema se vendía con una licencia que permitía modificarlo y redistribuirlo, a lo largo del tiempo fueron surgiendo una gran variedad de sistemas derivados del UNIX original. Los más conocidos, actualmente, son: Solaris, AIX, HPUX, SCO, BSD.

Por esta razón, varias veces se hizo necesario normalizar estos sistemas, para que cumplan con determinadas normas (POSIX, UNIX95, etc), para permitir la compatibilidad entre los diferentes sistemas.

De estas normas, el sistema operativo GNU/Linux satisface la norma POSIX-1, y casi completamente la POSIX-2.

1.2. Qué es GNU

La sigla GNU significa *GNU is Not Unix.*

En 1984, Richard Stallman fundó el Proyecto GNU con el objetivo de conseguir un sistema operativo libre y abierto. Esto es, un sistema operativo tal que los usuarios puedan usarlo, leer el código fuente, modificarlo, y redistribuirlo.

A partir de ese momento, un gran número de colaboradores se fueron sumando al proyecto, desarrollando software libre para reemplazar cada una de las herramientas del sistema UNIX.

La filosofía GNU apoya el crecimiento de la sociedad como un conjunto, haciendo especial hincapié en la valoración de las libertades personales, aún cuando esto puede estar en conflicto con intereses empresariales.

1.3. Qué es Linux

En 1991, Linus Torvalds completó el sistema con su kernel (que es la aplicación encargada de comunicar los procesos con el hardware de la computadora). A este kernel lo bautizó Linux.

De esta manera, se formó el sistema GNU/Linux.

1.4. Qué es BSD

La Universidad de Berkeley estuvo relacionada con el desarrollo de los sistemas operativos UNIX. Recibió de AT&T una versión gratuita de UNIX, y a partir de entonces comenzó a promover el desarrollo de aplicaciones para UNIX dentro de la universidad.

Mas adelante, desarrollo su propio sistema operativo UNIX, sin utilizar el código fuente de AT&T. El kernel fue creado desde Berkeley, pero las herramientas utilizadas son en su mayoría GNU, es decir las mismas que en el sistema GNU/Linux.

Existen actualmente 3 sistemas operativos libres, derivado de BSD: FreeBSD, OpenBSD y NetBSD.

1.5. Qué es X

El sistema operativo GNU/Linux cuenta con una interfaz gráfica, llamada XFree86 o simplemente X.

El protocolo X fue desarrollado por el MIT, principalmente como un logro académico para proporcionar un entorno gráfico a UNIX. La licencia mediante la cual se distribuye permite usarlo, modificarlo, redistribuirlo e incluso re-licenciarlo.

1.6. Qué son las distribuciones

El código fuente del sistema GNU y del kernel Linux está accesible a todo el mundo, sin embargo, hacer funcionar un sistema a partir del código fuente es bastante difícil. Por eso, un sistema operativo se distribuye (normalmente) en formato binario, es decir **ya compilado**.

Poco después de que apareciera el kernel Linux, comenzaron a aparecer las primeras distribuciones, que agrupaban versiones probadas de varios programas, junto con el kernel, de tal manera que formaban un sistema operativo listo para usar.

A medida que fue pasando el tiempo, algunas distribuciones se fueron haciendo más sofisticadas, otras desaparecieron, otras se hicieron comerciales y aparecieron mucha más. Existen distribuciones de muchos tipos: distribuciones que ocupan 1 diskette y distribuciones que llegan a ocupar 10 CDs; distribuciones orientadas a una finalidad en especial (redes, seguridad, etc) y distribuciones de uso general.

Cada usuario de GNU/Linux suele elegir la distribución con la que se siente más cómodo, y no tiene sentido entrar en discusiones acerca de cuál es mejor.

A menos que aclaremos lo contrario, lo que se enseña en este curso es aplicable a la gran mayoría de los sistemas UNIX, y a cualquiera de las distribuciones de GNU/Linux.

1.7. Qué son las plataformas

El mundo de las computadoras no se restringe a las Computadoras Personales con las que estamos en contacto todos los días. Existen diversas arquitecturas en las que una computadora se puede presentar. A cada una de estas arquitecturas la llamamos *plataforma*.

Ejemplos de algunas plataformas posibles son: iMac (de Macintosh), Sparc (de Sun), S/390 (de IBM), PlayStation (de Sony), XBox (de Microsoft).

En particular, la plataforma que utilizamos comúnmente se la denomina *i386*, ya que está basada en los procesadores de Intel, a partir del 386.

1.8. Por qué aprender acerca de GNU/Linux

Actualmente, a partir de la popularización de Internet, ha crecido en gran medida la cantidad de servidores de red en todo el mundo. Estos servidores deben contar con un sistema confiable, que pueda ofrecer los servicios de correo, web, etc. Por eso es que la gran mayoría de estos servidores utilizan algún derivado de UNIX como sistema operativo.

Dado que el sistema GNU/Linux cumple muy bien con estas características, con el adicional de que se puede obtener completamente gratis, año a año ha crecido el número de servidores que lo utilizan.

En el caso de las computadoras personales, a partir de la aparición de más y más aplicaciones gráficas, de uso general en el sistema GNU/Linux, se ha hecho posible la utilización de este sistema para computadoras de escritorio. La gran cantidad de software desarrollado en todo el mundo, permite tener muy buena compatibilidad con otras computadoras, y tener disponibles -al mismo tiempo- las herramientas UNIX.

1.9. Software Libre

A lo largo de todo este curso, siempre utilizamos Software Libre. Por lo general, este software lo hemos obtenido gratuitamente, sin embargo, debemos entender que el hecho de que el software sea libre está relacionado **con la libertad** que nos otorga a los usuarios de utilizarlo, modificarlo y distribuirlo, **no con el precio** al cual lo podemos obtener.

Al hablar de software libre se suelen clasificar los distintos grados de libertad a los que podemos tener acceso los usuarios.

Libertad Grado 0 El software se puede **usar**. Es la libertad que nos otorga casi cualquier software.

Libertad Grado 1 El software se puede **modificar**. Es decir, se puede personalizar, mejorar, adaptar para las necesidades particulares de un determinado usuario.

Libertad Grado 2 El software se puede **distribuir**. Es decir, se puede copiar, vender, prestar o compartir a las personas que el usuario desee, sin tener que pedir permiso al autor del software.

Libertad Grado 3 El software se puede **distribuir modificado**. Se trata de una suma de la 1 y la 2. Permite que las mejoras que un usuario le haya hecho a un determinado software puedan compartirse con otros usuarios.

Para poder considerar que una determinada aplicación cumple con los requisitos de software libre es necesario que estén dadas estas cuatro libertades para **cualquier** usuario.

En particular para poder tener la libertad de modificar el software, es necesario tener acceso al código fuente del programa en cuestión, y no solamente al código binario (llamamos código binario a aquél que es entendido por la computadora) ya que para poder modificar correctamente el software es necesario poder acceder al código fuente original.

Estas ideas de software libre como las conocemos hoy fueron desarrolladas y trabajadas durante mucho tiempo por Richard Stallman y sus seguidores, miembros de la “Free Software Fundation” (Fundación del Software Libre).

En general las ideas del software libre buscan promover la generación de mejor software (a través de la suma de los pequeños aportes de cada persona), y colaborar para que toda la sociedad se vea beneficiada con los avances del software. Es decir, logramos mejorar la sociedad al tener disponibles más y mejores herramientas.

Para más información sobre el software libre pueden visitar el sitio de internet del Proyecto GNU (http://www.gnu.org), que tiene una gran cantidad de documentos relacionados con la filosofía del software libre.

2. Conceptos Generales

2.1. Modo Gráfico - Modo Consola

Como ya dijimos anteriormente, GNU/Linux puede utilizar el *Modo Gráfico*, si utiliza la aplicación XFree86.

Por otro lado, llamamos *Modo Consola*, al modo que es puramente texto. Gran cantidad de los temas que se enseñan en este curso se pueden probar en modo consola, o bien en una consola gráfica, dentro del modo gráfico.

2.2. Comenzando la sesión

Dado que UNIX es un sistema multiusuario, para poder comenzar a utilizarlo debemos ingresar el usuario y password que nos identifica. Esto lo podemos hacer tanto en modo gráfico como en modo consola.

Muchas veces, cuando ingresemos nuestra contraseña a un sistema UNIX, no veremos los caracteres (ni siquiera un '*'). Esto se debe a que de esta manera es más difícil que un observador sepa la cantidad de caracteres que contiene nuestra contraseña.

Una vez que hayamos ingresado, tendremos acceso a nuestros archivos, y podremos ejecutar una gran cantidad de aplicaciones, según los permisos que se le hayan dado a nuestro usuario.

Es decir que, cada usuario que utilice el sistema tendrá un tratamiento distinto. A esto nos referimos cuando decimos que todo sistema derivado de UNIX es **multiusuario**.

En particular, existe un usuario que es el encargado de administrar el sistema, es el usuario que tiene acceso a los archivos de configuración, a instalar y desinstalar el sistema. Este usuario suele tener el nombre de *root*, o también *superusuario*.

2.3. Permisos y propiedad

En GNU/Linux los permisos sobre los programas y los archivos del sistema son bastante más específicos que en sistemas como Windows. Existen los conceptos de **propiedad** y **permisos**.

Propiedad: los usuarios son propietarios de archivos. Todos los archivos que un usuario crea son su propiedad.

Permisos: se refiere a lo que un usuario puede hacer con un determinado archivo, sea o no el dueño de ese archivo. Mediante los permisos, se controla quién puede visualizar, editar o ejecutar archivos en el sistema.

2.4. Consolas virtuales

En GNU/Linux y en sistemas UNIX modernos en general, es normal que nuestra computadora funcione como muchas terminales a un mismo tiempo. En un sistema UNIX tradicional, por otro lado, cada estación de trabajo nos daría acceso a una única terminal del sistema.

A esta característica que nos permite tener varias terminales en una sola computadora, la llamamos *consolas virtuales*.

Para cambiar de una terminal a otra, normalmente se utiliza la combinación de teclas **Alt-F1**, **Alt-F2**, etc. O bien, **Ctrl-Alt-F1**, **Ctrl-Alt-F2**, etc.

Dentro de una consola podemos movernos hacia arriba o hacia abajo, utilizando la combinación de teclas **Shift-PgUp** y **Shift-PgDn**.

2.5. Apagado del sistema

Como en otros sistemas operativos, es importante ejecutar un comando que cierre todas aquellas aplicaciones que estén corriendo, antes de apagar el sistema. En caso contrario, al reiniciar el sistema operativo se efectuará una verificación del disco rígido.

A diferencia de otros sistemas operativos, los sistemas UNIX están pensados para permanecer encendidos constantemente, esto hace que los usuarios comunes (los que no son el superusuario) no puedan apagar el sistema en forma remota.

En consecuencia, cuando solamente tenemos acceso de nivel usuario a una computadora con un sistema UNIX, la única manera de apagarla en forma segura es haciendo **Ctrl-Alt-Del** desde modo consola. Una vez presionada esta combinación de teclas, todos los procesos que se estén ejecutando serán detenidos, y la computadora se reiniciará.

En cambio, si fuéramos el superusuario del sistema, podremos ejecutar los comandos `halt` o `reboot` para apagar o reiniciar el sistema, respectivamente.

3. Recorrido por el sistema

Vamos a ver algunos comandos básicos de todo UNIX, que nos permitirán familiarizarnos con el sistema. Para poder utilizar estos comandos ya debemos haber iniciado la sesión.

Cuando hemos iniciado la sesión estaremos delante de un *prompt* (solicitud), que es la línea de comandos de UNIX. El formato de este prompt será similar a: `user@anteojito:~$`.

La mayoría de estos comandos son herramientas simples, que realizan una sola tarea. Más adelante aprenderemos como combinar lo que hacen dos o más herramientas para lograr un resultado más interesante.

Algunos de estos comandos pueden recibir opciones o parámetros, que pueden hacerlos realizar tareas diferentes. En UNIX, casi todas las opciones que reciba un comando estarán precedidas por el caracter `-`, y pueden combinarse según sea necesario.

Es importante notar que UNIX es sensible a las mayúsculas y minúsculas (*case sensitive*), de forma que si queremos obtener la salida correcta es necesario escribir los comandos que aquí se explican tal cual se presentan (casi siempre en minúsculas).

3.1. Comandos Sencillos

3.1.1. date

Ejecutando el comando `date` obtendremos la fecha y la hora actual. También es el comando que se usa para cambiar la fecha de la máquina.

3.1.2. who

El comando `who` nos permite ver los usuarios que están utilizando el sistema, incluyendo la fecha en la que se conectaron al sistema.

Además, una versión alternativa (`who am i`) nos muestra únicamente nuestra conexión.

3.1.3. uptime

Podemos ver cuánto tiempo hace que se ha iniciado el sistema de nuestra computadora ejecutando el comando `uptime`. También obtendremos alguna información adicional, como la cantidad de usuarios que están utilizando el sistema, o la carga promedio que tiene el procesador.

3.1.4. clear

El comando `clear` sirve para limpiar la pantalla. Es equivalente al `cls` de DOS.

3.1.5. echo

`echo` es un comando muy sencillo. Lo que hace es repetir todo lo que recibe por línea de comandos. Si ejecutamos: `echo Hola` obtendremos la salida `Hola`.

A pesar de parecer inútil, este comando nos será de bastante utilidad cuando aprendamos más sobre el intérprete de comandos del sistema.

3.2. Comandos relacionados con archivos y directorios

3.2.1. pwd

El comando `pwd` es muy sencillo, nos muestra la ruta de directorios en la que estamos situados en este momento. Por ejemplo, **/home/user**.

3.2.2. ls

Para listar los archivos dentro de un determinado directorio utilizaremos el comando `ls`. El equivalente en DOS es 'dir'.

Si ejecutamos `ls` sin ninguna opción, nos listará los archivos del directorio actual. Si, en cambio, ejecutamos `ls directorio`, nos listará los archivos de ese directorio.

Puede recibir varias opciones:

- `-l`: muestra mucha más información sobre los archivos, como el usuario y el grupo al que pertenece, el tamaño, los permisos, etc.
- `-a`: muestra todos los archivos, incluyendo los ocultos.
- `-t`: ordena los archivos por fecha de modificación.

Estas opciones pueden combinarse, utilizando un solo signo '-', por ejemplo: `ls -lat`.

3.2.3. touch

El comando `touch archivo` puede tener dos consecuencias: si `archivo` no existe aún, lo crea con tamaño 0 y como propiedad de nuestro usuario. Por otro lado, si `archivo` ya existe, actualiza la fecha de modificación.

3.2.4. cp

El comando `cp` es el que se utiliza para copiar archivos.

Si escribimos `cp viejo nuevo`, copiaremos el archivo `viejo` con el nombre `nuevo`. Es decir, el archivo origen se escribe primero y a continuación el archivo que se va a crear. Una vez hecha la copia, tendremos dos archivos diferentes, con el mismo contenido.

Por otro lado, también podemos ejecutar `cp archivo1 archivo2 directorio`, de forma que los archivos `archivo1` y `archivo2` se copiarán dentro de `directorio`.

3.2.5. mv

Muy similar a `cp`, el comando `mv` es el que se utiliza para mover archivos de un lugar a otro, o para cambiarle el nombre a un archivo. Si ejecutamos, `mv viejo nuevo`, el archivo `viejo` habrá pasado a llamarse `nuevo`.

Por otro lado, si ejecutamos `mv archivo1 archivo2 directorio`, los archivos `archivo1` y `archivo2` se moverán dentro de `directorio`.

3.2.6. rm

Para borrar archivos utilizamos el comando `rm`. Hay que usarlo cuidadosamente, porque una vez que los archivos han sido borrados, no pueden recuperarse **de ninguna forma**.

Si deseamos que `rm` nos pregunte si queremos borrar o no un archivo, debemos utilizar la opción `-i`, mientras que si deseamos que no nos pregunte utilizamos la opción `-f`. Dependerá de la configuración del sistema cual de estas dos opciones es la que está seleccionada por omisión.

3.2.7. mkdir

Utilizamos el comando `mkdir directorio` para crear directorios. Pueden utilizarse rutas absolutas o relativas. Es decir que si queremos crear el directorio `/home/user/temp`, y estamos situados dentro del directorio `/home/user`, podremos ejecutar `mkdir temp` o `mkdir /home/user/temp` indistintamente.

3.2.8. rmdir

Para borrar directorios utilizamos el comando `rmdir directorio`. Solamente funcionará si el directorio está vacío. En caso contrario, habrá que borrar primero los archivos, para luego borrar el directorio.

3.2.9. cd

El comando `cd` nos permite cambiar de directorio, igual que en DOS.

Por ejemplo, `cd /` nos lleva al directorio raíz, que es de donde surgen todos los directorios del sistema.

Para cambiar a otro directorio dentro del árbol, podemos utilizar `cd usr`, o `cd /home/user`. Más adelante veremos cómo se organiza el árbol de directorios, y qué hay en cada uno.

Utilizado sin ningún otro parámetro, `cd` nos lleva al directorio personal del usuario (*home*). Otra manera de ir al directorio personal es utilizar `cd ~`, ya que el símbolo `~` identifica al directorio de cada usuario.

Para cambiar al directorio padre del directorio actual se utiliza `cd ..` (no olvidar el espacio). Mientras que para volver al directorio en el que nos encontrábamos antes de ejecutar el último `cd`, podemos ejecutar `cd -`.

3.2.10. file

En UNIX los archivos no se identifican por su extensión, como en DOS. Se les puede poner una extensión, pero es de adorno.

El comando `file` nos permite tener una idea del contenido de un archivo. Al ejecutar `file archivo`, inspecciona partes del archivo para darse cuenta qué clase de archivo es. Puede decirnos que se trata de un archivo de texto, un directorio, una imagen, etc.

3.2.11. du

El comando `du`, *Disk Usage*, nos muestra el espacio que ocupan todos los directorios a partir del directorio actual. El número de la primera columna es el espacio ocupado por el directorio y está expresado en kb.

- `du -s` nos muestra únicamente el total.
- `du -a` muestra lo que ocupan los archivos, además de los directorios.
- `du -h` hace el listado, indicando la unidad (human readable).
- `du archivo` nos dice cuánto ocupa el archivo.

3.2.12. df

El comando `df`, *Disk Free*, nos muestra qué tan ocupados están los discos del sistema. El parámetro `-h`, al igual que en el caso del comando `du`, nos muestra las unidades en que se miden los numeros presentados.

3.2.13. find

El comando `find` permite encontrar archivos, utilizando diversas técnicas. En principio, si se le pasa como parámetro únicamente una determinada ruta, por ejemplo `find /home/user`, el comando buscará todos los archivos y directorios que se encuentren a partir de esa ruta.

Utilizando algunos otros parámetros es posible buscar los archivos por diversos criterios.

- `find . -name "hola.txt"` encuentra todos los archivos llamados `hola.txt` que se encuentren a partir del directorio actual. Las comillas no son obligatorias, pero son recomendables si se quieren usar opciones más complejas (por ejemplo, utilizando meta-caracteres de shell, que se explican en la sección 4.1).
- `find . -size 50k` busca los archivos que ocupan 50 kilobytes a partir del directorio actual. Si se utiliza `find . -size 20c`, buscará los archivos que ocupen 20 bytes. Y si se utiliza `find . -size 5b`, buscará los archivos que ocupen 5 bloques de 512 bytes cada uno.
- `find /home/user -empty` busca todos los archivos que se encuentran vacíos, a partir del directorio `/home/user`.

Es posible, además, utilizar opciones adicionales para la búsqueda, que indiquen la profundidad de la búsqueda, que agreguen otros criterios adicionales a los explicados, o que indiquen una acción a llevar a cabo una vez encontrados los archivos.

3.2.14. locate

Otro comando que nos permite buscar archivos es `locate`. Las busquedas de `locate` nos sirven para encontrar archivos en cualquier parte del sistema, por ejemplo `locate bash` nos va a mostrar todos los archivos y directorios del sistema que en su nombre contienen la palabra **bash**.

Las busquedas de `locate` son mucho más rápidas que las de `find`, esto se debe a que utiliza una base de datos que contiene los nombres de los archivos de todo el sistema. Esta base de datos no se actualiza constantemente, por lo que `locate` no encontrará los archivos recién agregados al sistema e incluirá los recientemente removidos hasta que esta base de actualice.

Para actualizar esta base de datos debemos ejecutar `updatedb`, como superusuario. En las máquinas que están prendidas constantemente, este programa se ejecuta una vez al día, durante la madrugada, de manera que la base de datos permanece acualizada.

3.2.15. cat

Ejecutando `cat archivo` podremos ver el contenido de `archivo`. Este comando puede recibir una serie de archivos, y el resultado será que nos mostrará un archivo a continuación del otro.

Un caso especial se produce cuando ejecutamos `cat` sin ningún nombre de archivo. En este caso, el comando esperará a que nosotros le demos una entrada, y la irá reproduciendo línea por línea. Hasta que presionemos la combinación **Ctrl-d**, que indica que la entrada ha terminado.

3.2.16. od

El comando `od` *Octal Dump*, nos permite ver byte a byte el contenido de un archivo. La primera columna es la dirección de cada línea que vemos. Utilizando las distintas opciones, podemos visualizarlo en varios formatos:

- `od archivo` nos muestra el contenido del archivo expresado en números octales, generalmente tomados de a dos bytes.
- `od -b archivo` nos muestra el contenido, en números octales, byte a byte.
- `od -c archivo` nos muestra los caracteres que forman el archivo, uno por uno.
- `od -cb archivo` nos muestra los caracteres, y debajo de cada caracter el número octal del byte.
- `od -h archivo` nos muestra el contenido, en números hexadecimales, tomados de a dos bytes.

Las nuevas versiones de `od` soportan muchos más formatos, utilizando la opción `-t formato`.

3.2.17. wc

El comando `wc archivo`, se utiliza para contar la cantidad de líneas, palabras y letras que tiene un archivo.

3.2.18. less

El comando `less` permite paginar la salida de otros comandos, o bien, el contenido de algún archivo.

Si ejecutamos `less archivo`, veremos la primera página del archivo. Si este archivo es lo suficientemente largo, podremos movernos hacia abajo y hacia arriba utilizando PageUp, PageDown, Home, End, Enter, los cursores, la barra espaciadora, etc.

También podemos realizar búsquedas dentro del archivo, para ello utilizamos la barra `/`, seguida del patrón que queremos buscar. Por ejemplo, si tipeamos `/consola`, nos mostrará la primera ocurrencia del patrón `consola`. Para ver la siguiente ocurrencia, utilizamos `n`, y para ver la ocurrencia anterior `N`.

Para salir, utilizamos `q`.

3.3. Comandos relacionados con procesos

Cada aplicación que se ejecuta dentro de un sistema UNIX es un proceso. Algunos procesos están ejecutándose sin que nosotros lo sepamos. Otros procesos comienzan a ejecutarse cuando nosotros se lo indicamos.

Cada proceso que se ejecuta dentro de nuestra computadora tiene un número que lo identifica, llamado *Process ID* (PID). Este número será el que deberemos utilizar para referirnos a ese proceso con otros comandos.

Vemos a continuación los comandos básicos para manejar procesos dentro de los sistemas UNIX.

3.3.1. top

El comando `top` nos muestra algunos de los procesos que se están ejecutando, ordenados por el tiempo de procesador de la computadora que consumen. Muestra algunos datos adicionales de cada proceso, por ejemplo, en la primera columna, podemos observar el PID de cada uno.

Además, muestra otros datos acerca del uso que se le esta dando a la máquina.

Para salir: q.

3.3.2. ps

El comando `ps` nos muestra la lista de procesos que se están ejecutando en la computadora. En particular, es interesante ver la salida de `ps ax`, que nos muestra todos los procesos, tanto los de nuestro usuario como los de otros, e incluso los procesos que no tienen usuario.

La primera columna de la salida de `ps` también contiene el PID del proceso.

3.3.3. Ctrl-z

La combinación de teclas **Ctrl-z** sirve para suspender una tarea dentro de su ejecución.

Por ejemplo, si estamos ejecutando el proceso `top` y presionamos **Ctrl-z**, se suspenderá la ejecución de `top` y volveremos a obtener la línea de comandos. Antes de devolvernos la línea de comandos, nos indicará cuál es el número de trabajo del proceso que hemos suspendido.

Podemos iniciar varias tareas distintas, y luego suspenderlas. De forma que el número de trabajo de las tareas suspendidas se va incrementando.

3.3.4. bg - fg

El comando `bg` permite que el proceso que se halle suspendido, continúe ejecutándose en *background* (de fondo). Mientras que el comando `fg` permite que un proceso suspendido pase a *foreground* (a la pantalla principal). Por ejemplo, `find /usr -name nethack` busca dentro de /usr (grande) un archivo llamado nethack, esto puede tomar una considerable cantidad de tiempo, apretando `Ctrl-z` y luego ejecutando `bg` conseguimos que se siga ejecutando de "fondo", mientras nosotros podemos seguir usando la consola. Cabe aclarar que cuando find encuentre algo lo imprimira en la pantalla.

3.3.5. jobs

Para poder ver qué comandos se están ejecutando en background y cuáles han sido suspendidos (o detenidos), podemos utilizar el comando `jobs`.

La lista que nos muestra este comando incluye el número de trabajo, que no es el mismo que el PID, y podemos utilizar este identificador para decidir cuál de las tareas pasar a foreground o background.

Por ejemplo: `fg 1` pasa a foreground el trabajo número 1. Mientras que `bg 3` pasa a background el trabajo número 3.

3.3.6. kill

Este comando nos sirve para interrumpir la ejecución de un determinado proceso. El comando envía una señal al proceso por la cual (normalmente) se cierra.

Podemos ejecutarlo teniendo en cuenta el PID del proceso. Por ejemplo: `kill 1234`, matará el proceso 1234. O bien, teniendo en cuenta el número de trabajo. En ese caso: `kill %2`, detendrá el trabajo número 2.

3.3.7. Ctrl-c

Cuando una aplicación se está ejecutando en foreground, y deseamos detenerla, podemos utilizar la combinación de teclas **Ctrl-c**.

El uso de esta combinación es equivalente a ejecutar el comando `kill` con el número de proceso de la aplicación que estamos utilizando.

3.4. Obteniendo más información

En GNU/Linux existen muchísimos documentos de ayuda, manuales y HOWTOs, que están pensados para que cualquier persona pueda encontrar información sobre lo que está buscando.

3.4.1. man

Un comando muy importante es `man`. Este comando nos mostrará las hojas del manual del programa que estamos queriendo buscar.

Por ejemplo, `man date` nos mostrará el manual del comando `date`, que ya sabemos que sirve para ver y configurar la fecha, aquí está explicado cómo utilizarlo.

Podemos movernos dentro de las páginas de los manuales utilizando la barra espaciadora, Enter, los cursores y el mismo sistema de búsqueda que utilizamos en `less`. Para salir, utilizamos `q`.

3.4.2. info

Un comando muy similar a `man`, es el comando `info`. Las páginas que nos muestra este comando suelen tener una mayor cantidad de información acerca de la aplicación sobre la cual estamos consultando.

Por ejemplo `info sh-utils`, contiene información detallada sobre algunas de las utilidades del intérprete de comandos (*shell*), que se verá más adelante.

3.4.3. help

Algunos comandos (como `fg`), son parte interna del intérprete de comandos, y por esta razón no tienen una página del manual que los explique.

Para saber de qué manera utilizar estos comandos, usamos `help`. La ayuda que nos da este comando es más sintética que la de `man`.

Por ejemplo `help jobs`, nos informará sobre el uso del comando `jobs` visto anteriormente.

3.4.4. Archivos con información

Dentro del directorio `/usr/share/doc`, encontramos una gran cantidad de documentos que tratan las distintas aplicaciones que tenemos instaladas en nuestro sistema.

En particular, el directorio `/usr/share/doc/HOWTO`, contiene artículos sobre cómo hacer determinadas cosas dentro de nuestro sistema.

4. Shell

El shell es el intérprete de comandos. En DOS normalmente el shell es el command.com, en UNIX existen muchos shell usados habitualmente.

sh Desde la séptima edición de UNIX el shell por excelencia es el **sh**. Fue escrito por Steven Bourne, y es por eso que se lo suele llamar *Bourne Shell*. Está disponible en todas las versiones de UNIX y es lo suficientemente básico como para que funcione en todas las plataformas.

csh Un shell un poco mejor con respecto al **sh** es el **csh**, que fue escrito por Bill Joy, y debe su nombre al lenguaje de programación C. Al hacer scripts en este shell puede utilizarse una sintaxis similar a la de C.

ksh Otro shell, que como ventaja maneja un historial de comandos, es el **ksh** (korn shell). Está basado en **sh**, con algunos agregados muy básicos para hacerlo más amigable.

bash Uno de los shell más avanzados, muy popular en la comunidad GNU/Linux es el **bash**. El nombre significa *Bourne Again Shell*. Tiene licencia GNU y se suele incluir como shell predeterminado en las distribuciones.

Ofrece las mismas capacidades que **csh**, pero incluye funciones avanzadas, tanto para el usuario como para el programador.

En particular, podremos acceder a un historial de los comandos ejecutados, que se conserva incluso al pasar de una sesión a otra, utilizando los cursores.

Además, completa los nombres de comandos y archivos automáticamente, al presionar la tecla TAB.

Hay muchas otras versiones de shell además de éstas. Pero el estilo general de todos es muy similar. Lo que estudiemos en este curso será compatible con la mayoría de ellos.

4.1. Meta-caracteres

Además de ejecutar los comandos que nosotros le indicamos, el shell interpreta ciertos caracteres especiales, a estos caracteres los llamamos metacaracteres.

Cuando nosotros utilizamos algún meta-caracter, los comandos no lo reciben, sino que el shell lo reemplaza por lo que corresponda, y le pasa al comando ejecutado el resultado de ese reemplazo.

Eso es lo que entendemos por *interpretar*: reemplazar el caracter por otro caracter o por una cadena de caracteres, según corresponda.

4.1.1. Meta-caracteres relacionados con archivos

`*` Cuando el shell encuentra un `*`, lo reemplaza por una lista de los archivos que concuerdan con la expresión indicada.
Por ejemplo: `echo *` nos mostrará todos los archivos del directorio. `echo a*` nos mostrará todos los archivos del directorio que comiencen con a. `echo *o` nos mostrará todos los archivos que terminen con o. `echo /usr/local/*` nos mostrará todos los archivos que estén en ese directorio.
En el caso de que no hubiera ningún archivo que concuerde con la expresión, generalmente, nos mostrará la expresión que hayamos escrito.

`?` Al encontrar un `?` el shell lo reemplaza por cualquier otro caracter. Es decir que la expresión que escribamos se reemplazará por todos los archivos que en esa posición tengan cualquier caracter, y en el resto de la cadena tengan lo que hemos escrito.
Por ejemplo: `echo ?ola` nos podría mostrar archivos como `hola`, `sola`, `Pola`. `echo a??a`, podría mostrar `allá`, `arca`, `Asia`.
Al igual que con el `*`, si ningún archivo concuerda con el patrón, generalmente, nos mostrará la misma expresión que hemos escrito.

`[ ]` Encerrados por los corchetes, podemos escribir un rango de caracteres con los cuales queremos que el shell concuerde.
Por ejemplo, `ls [af]*` nos mostrará todos los archivos que comienzan con a o con f.
Podemos además especificar un rango de caracteres, con un guión en el medio. Por ejemplo, `a-z` (letras minúsculas), `0-9` (números), etc. y combinarlos con caracteres individuales siempre que no sea ambigua la interpretación. (Considerar la concordancia con el caracter -).
Por ejemplo, podemos querer sólo los archivos que comienzan con números seguidos de un -, en ese caso escribiríamos `ls [0-9]-*` o `ls [0-9][0-9]-*`, si comienzan con dos números seguidos de un -.

`[^ ]` Cuando al comienzo de la cadena que está encerrada por los corchetes encontramos el caracter `^`, estamos indicando que debe concordar los caracteres que no se encuentran en el rango.
Por ejemplo, `ls [^0-9]*`, nos listará todos los archivos que no comiencen con un número.

4.1.2. Meta-caracteres relacionados con comandos

Ejecutar un comando es tan sencillo como escribir el comando y apretar ENTER. Sin embargo, utilizando algunos de los meta-caracteres de shell podemos combinar los comandos entre sí, y lograr resultados mucho más importantes.

; El `;` es un separador de comandos, nos permite ejecutar un comando a continuación de otro, equivalente a lo que sucedería si ejecutáramos primero uno, y al terminar ejecutáramos el siguiente.

Es decir si escribimos `ls; echo Hola` veremos la salida del `echo` a continuación de la del `ls`.

( ) Los paréntesis sirven para encerrar grupos de comandos, y tratarlos como si fueran uno solo.

& El `&` manda el comando a background, esto quiere decir, que nos devuelve la línea de comandos inmediatamente después de apretar Enter, mientras el comando sigue ejecutándose en segundo plano.

La ejecución de tareas en segundo plano ya se ha estudiado anteriormente, cuando se vieron los comandos relacionados con procesos. Este meta-caracter funciona de manera equivalente, y sus resultados pueden corroborarse utilizando el comando `jobs`.

Para ver un ejemplo, vamos a usar un nuevo comando, `sleep`, (un comando simple que espera una determinada cantidad de segundos). Por ejemplo `sleep 5`, espera 5 segundos antes de devolvernos la línea de comandos.

Ahora, utilizando `&`: `(sleep 20; echo Hola) &`. Al escribirlo nos mostrará el PID del comando que estamos ejecutando, y nos devolverá el shell; 20 segundos después, veremos aparecer "Hola" en nuestra línea de comandos.

Antes de que termine de ejecutarse, podemos ejecutar `jobs` y observar que el proceso se está ejecutando, o bien `ps` y observar que el comando en ejecución es **sleep**.

Ejercicio: hacer un comando equivalente que nos avise que ya pasaron los cinco minutos necesarios para que hierva una pava de agua.

Además, el `&` nos puede servir para separar comandos: cada vez que lo utilizamos para separar comandos, mandará al comando que esté a su izquierda a background.

4.1.3. Otros meta-caracteres

'...' Al encontrar una cadena encerrada entre ' ', el shell tomará el contenido de la cadena literalmente, es decir, sin interpretar los meta-caracteres contenidos en ella.

Por ejemplo, **echo '* ?* [A-Z-]*'** nos mostrará * ?* [A-Z-]*.

Notar que si no cerramos las comillas y apretamos ENTER, el shell nos mostrará una línea en blanco esperando que sigamos ingresando nuestro comando, hasta que cerremos las comillas.

**** Utilizamos una \\ para escapar el siguiente caracter. Escapar significa que el shell no lo interpretará como un meta-caracter.

Por ejemplo `echo \*` nos mostrará un `*`.

El `#` es el señalador de comentario. Si el shell encuentra un # al comienzo de una palabra, descartará todos los caracteres hasta en final de línea. Por ejemplo, `echo 3.1416 # Pi con un error de 0.0` nos mostrará únicamente `3.1416`.

4.1.4. Entrada y Salida

UNIX tiene un extenso manejo de entrada y salida, es una de las características principales que nos permite combinar pequeñas herramientas para lograr resultados más complejos.

La mayoría de los comandos UNIX que nosotros utilizamos tienen una entrada estándar, una salida estándar y una salida para errores estándar. Las denominamos **stdin**, **stdout** y **stderr** respectivamente.

La entrada estándar por omisión es el teclado, mientras que la salida estándar y la salida de errores son, por omisión, la pantalla.

Un comando genérico, lee datos de la entrada estándar, los procesa de alguna manera, y luego emite el resultado por la salida estándar. En el caso de que durante el proceso hubiera algún error, emitirá una aviso de ese error por la salida de errores.

El Shell se encarga de relacionar estos tres, lo cual no impide que un determinado programa maneje su entrada y su salida de una manera diferente.

$>$ El caracter `>` nos permite direccionar la salida estándar de un comando a un archivo. De manera que `ps ax > procesos` guardará en el archivo `procesos` la salida del ps.

$<$ El caracter `<` nos permite direccionar la entrada estándar de un comando desde un archivo. Por ejemplo, el comando `mail` nos sirve para mandar mensajes a otros usuarios, si escribimos `mail user < archivo` mandará un mensaje con el contenido del archivo al usuario user.

$>>$ Usar un `>>` en lugar de un `>` nos permite direccionar la salida estándar a un archivo, sin sobreescribirlo, sino que le agrega los datos que nosotros queramos al final. Si ahora hacemos `ps ax >> procesos` tendremos el listado de procesos dos veces en un mismo archivo.

2$>$ Utilizar `2>` nos permite redirigir la salida de errores a un archivo. Por ejemplo, si ejecutamos `ls archivo-feo 2> test`, el error del `ls`, indicándonos que el archivo-feo no existe se almacenará en test.

| Para relacionar la salida estándar de un comando, con la entrada estándar de otro comando, utilizamos el caracter `|`. Por ejemplo, podemos relacionar la salida de `ls` con la entrada de `wc`. Haciendo `ls | wc`, la salida de este comando será la cantidad de líneas, palabras y caracteres que produjo `ls`.

Este comando recibe el nombre de *pipe*, que en inglés significa cañería o tubería. Es decir que es un comando que *entuba* la salida de un comando con la entrada de otro.

Es interesante observar lo que sucede cuando hacemos: `ls > nuevo-archivo`, esto es, el archivo nuevo aparece dentro del listado que hace ls. Esto se debe a que el shell, al hacer la relación entre el archivo y el comando, crea el archivo, y luego llama al `ls`.

Además es necesario tener en cuenta que un comando no puede utilizar como entrada y salida un mismo archivo. Por ejemplo, al ejecutar `cat archivo > archivo`, el intérprete de comandos nos indicará que esto no es posible.

4.1.5. Ejercicios

1. Explicar en qué se diferencian `ls *` y `echo *`.
2. Explicar en qué se diferencian `ls /` y `echo /`.
3. Crear un archivo que contenga la cantidad de archivos en un directorio.
4. Crear dos archivos: `a.txt`, que contenga `hola`, y `b.txt`, que contenga `chau`. Luego concatenarlos en un archivo `ab.txt`.

5. Manejo de Archivos

Una de las características de UNIX es que todo es un archivo. Esto no es una simplificación, el sistema representa los dispositivos (disco rígido, placa de sonido, CDs, disqueteras, algunas conexiones de red, etc) que están conectados a él como archivos.

Estos archivos están todos ubicados a partir del directorio raíz, a partir del cual se abre un árbol de directorios. Al directorio raíz se lo identifica como /. Si hacemos `ls /` veremos los directorios que se abren a partir del raíz.

Una de las ventajas del manejo de archivos de UNIX es que los usuarios no necesitan saber dónde está cada directorio. Cuando hacemos `ls /` puede suceder que algunos directorios estén en el disco rígido de la PC, otros estén en un servidor de la red, otros en algún otro servidor, y nosotros no nos enteraremos.

Identificamos con el nombre de **file-system** a una estructura jerárquica de directorios y archivos. Es decir, una determinada colección de archivos. En un solo sistema UNIX pueden estar *montados* varios file-systems diferentes. Esto es, un solo sistema UNIX puede agrupar los archivos que están almacenados en distintas particiones de un disco rígido, en una disquetera, en un CD, etc.

El árbol de directorios que vemos a partir del directorio raíz, es la forma en la que organizamos los archivos, no necesariamente el lugar donde los archivos están ubicados físicamente.

5.1. Montar y desmontar file-systems

Utilizando el comando `mount` veremos los file-systems que tengamos montados. La salida de este comando nos lista: la identificación del file-system, el lugar en el árbol de directorios en el que está montado y el tipo de file-system que constituye. Además, entre paréntesis, podemos ver los flags con los que se ha montado.

Cuando queremos leer de un diskette, deberemos montarlo, ya que el diskette constituye una nueva estructura de archivos. Podremos montar diskettes que estén grabados en distintos formatos, por ejemplo msdos, ext2, etc.

Para montar el diskette utilizaremos `mount /floppy`, ya que /floppy es el directorio en el que está configurado que se debe montar el dispositivo de la disquetera, como un tipo de file-system automático, es decir que detecta automáticamente qué tipo de filesystem se tiene.

Es necesario desmontar el diskette antes de sacarlo, ya que -si no lo hacemos- el sistema puede no haber terminado de actualizar los cambios hechos al file-system del diskette. Para desmontarlo utilizaremos el comando: `umount /floppy`.

Otro comando relacionado con los file-systems que se encuentran montados en el sistema es `df`. Este comando nos muestra el espacio que está siendo utilizado y el que todavía está libre, en los file-systems que se encuentren montados.

5.2. Árbol de directorios

Existen diferentes maneras de ordenar la información dentro de los directorios. Veremos un orden básico que la mayoría de las distribuciones utilizan.

/bin contiene los archivos ejecutables básicos del sistema. Muchos de estos comandos ya los usamos o los vamos a usar en el futuro.

En este directorio, por ejemplo, encontramos algunos de los comandos de shell que nombramos.

/dev contiene los archivos que representan a dispositivos. Los archivos que se encuentran en este directorio están relacionados con periféricos de la máquina, por ejemplo: /dev/fd0 es el archivo que representa la primera disquetera, /dev/hda representa al primer disco rígido IDE, /dev/hda1 representa la primera partición del primer disco rígido IDE, /dev/tty representa la consola actual que se está usando. Así: `echo hola > /dev/tty` nos mostrará la misma salida que un simple `echo hola`.

Estas identificaciones corresponden al sistema de archivos utilizado por las distribuciones de GNU/Linux. En otros sistemas UNIX (como BSD, Solaris, etc). Las identificaciones serán distintas, pero las podremos encontrar en el mismo directorio /dev.

Un dispositivo curioso es /dev/null. Se trata de un dispositivo que borra todo aquello que se mueva a él, como un agujero negro. Otro dispositivo interesante es /dev/random, que nos muestra "basura random", es decir cualquier tipo de caracteres, generados al azar según los procesos que se estén ejecutando en la máquina y la entrada del usuario.

/etc contiene los archivos de configuración del sistema y de muchos de los programas instalados en el sistema. Además, contiene los scripts de inicio que se ejecutan cuando arranca la máquina. Generalmente los archivos que se encuentran en este directorio pueden ser editados sólo por el administrador de la máquina, es decir el superusuario, normalmente llamado "root".

/home en este directorio se encuentran los directorios principales de los usuarios. En estos directorios los usuarios tienen permisos de leer, escribir y ejecutar según deseen.

/lib contiene las librerías necesarias para ejecutar los comandos que se encuentran en /bin y /sbin. Las librerías son rutinas que los programas utilizan frecuentemente, y pueden ser compartidas por varios programas al mismo tiempo.

Esto se debe a que las librerías no están incluidas dentro de los programas, para que sea sencillo reusar el código, y aprovechar mejor el espacio.

En otros sistemas UNIX, las librerías pueden estar incluidas en los comandos, con lo cual este directorio no es necesario.

/sbin contiene los archivos ejecutables que son necesarios para poder administrar el sistema.

/usr contiene archivos que serán utilizados una vez que el sistema ya está funcionando. No son imprescindibles para el funcionamiento del sistema.

Por dentro se subdivide nuevamente en un árbol muy parecido al del directorio raíz.

Encontramos, además, directorios como **src** (donde se suelen guardar los archivos con el código fuente del sistema), **games** (que tiene juegos), **X11R6** (que contiene el sistema X), **share** (que contiene archivos a los que pueden acceder todos los programas y todos los usuarios).

5.3. Los archivos por dentro

El contenido de un archivo común no es mas que una sucesión de bytes, pero cómo está esta información almacenada y cómo accedemos a ella es más interesante. Para tener una idea de cómo se guarda la información en los sistemas UNIX veremos el manejo de *inodos*.

5.3.1. Inodos

Un inodo contiene el tamaño de un archivo, las tres fechas relacionadas con el archivo (modificación del contenido, acceso al contenido y modificación del inodo), los permisos, el dueño, el grupo, el número de links, e información sobre dónde están físicamente los datos (los bytes) del archivo en el disco.

Notemos que el nombre del archivo no está guardado en el inodo.

De esta forma, un directorio es simplemente un archivo que contiene una lista de números de inodos y nombres de archivos.

5.3.2. Links

Dentro del inodo se guarda el número de links del archivo. Un link es una entrada en un directorio, es decir en la lista de inodos y nombres de archivos. Un archivo puede tener muchos links. Esto significa que dos archivos con distinto nombre pueden apuntar a un mismo inodo, y por lo tanto tener el mismo contenido. Son el mismo archivo con dos nombres diferentes.

Es importante recalcar que una vez creado el link, es imposible decir cuál es el nuevo archivo y cuál es el que ya estaba creado. Si borramos uno de estos archivos, no se borrará el contenido del archivo, sino únicamente la entrada de directorio, y se decrementará la cantidad de links al inodo. Solamente se eliminará el contenido del archivo cuando el número de links llegue a cero.

A este tipo de links se los llama *hard links*, existe otro tipo de links, llamado *soft links* o también *symlinks* (Symbolic Link). Estos links no comparten el inodo del otro archivo, ni aumentan la cantidad de links que tiene el otro archivo, porque se trata únicamente de un puntero a otro archivo, que se puede encontrar en cualquier parte del sistema de archivos. Es muy similar al *Acceso Directo* de Windows.

Dado que los datos de los inodos se guardan por separado en cada dispositivo (es decir, partición de disco, CD, floppy, etc), no es posible crear *hard links* entre dispositivos diferentes. Sin embargo, sí es posible crear *symlinks*.

5.3.3. ln

El comando `ln` nos permite crear tanto *hard links* como *soft links*.

Para crear un hard link deberemos escribir `ln viejo nuevo`. De tal manera que `nuevo` estará asignado al mismo inodo al que ya estaba asignado `viejo`. Si hacemos `ls -li` del directorio actual, podemos ver el número de inodo en la primera columna, y el número de links en la tercera. veremos que tanto `viejo` como `nuevo` tienen el mismo número de inodo y de links.

Para crear un *symlink* utilizamos `ln -s viejo nuevo`. En este caso, al hacer `ls -li` vemos que `nuevo` tiene un inodo diferente a `viejo`, y que en el nombre aparece la ruta a la que apunta.

5.4. Permisos

Como ya dijimos antes, todos los archivos tienen un dueño y un grupo. Para ver el dueño y el grupo, podemos hacer `ls -l`, la tercera columna es el usuario y la cuarta el grupo.

Analizaremos la primera columna (que puede parecer un jeroglífico, pero lo que nos indica es los permisos que tiene cada archivo).

El primer caracter nos indica qué tipo de archivo es. Si tiene una `"d"` es un directorio, si tiene una `"l"` es un symlink, si tiene un `"-"` es un archivo común.

Los siguientes tres caracteres indican los permisos que tiene el archivo para el dueño. La primera columna indica lectura, y puede tener una `"r"` si está habilitado el permiso o `"-"` si no lo está. La segunda indica permiso de escritura, y puede tener una `"w"` si está habilitado o un `"-"` si no lo está, la tercera indica ejecución, y puede tener una `"x"` si está habilitado o un `"-"` si no lo está.

Los otros tres caracteres son los permisos para el grupo, y los últimos tres son los permisos para el resto de los usuarios.

En el caso de los directorios, el permiso de lectura significa que se puede listar el contenido del directorio, haciendo -por ejemplo- `echo dir/*`, el permiso de escritura significa que se pueden crear y borrar archivos en el directorio y el permiso de ejecución significa que se pueden buscar archivos, hacer un `cd`, etc.

5.4.1. chmod

Para cambiar los permisos de un archivo utilizamos el comando `chmod`. Solamente el dueño de un archivo (o el administrador del sistema) puede cambiarle los permisos a un archivo. Aún si no tiene ninguno de los permisos explicados, el dueño del archivo siempre tiene permiso de cambiar los permisos de un archivo.

Hay dos maneras diferentes de utilizarlo. Podemos sumar o restar permisos a los que ya están, o bien configurar todos los permisos a la vez.

Si queremos sumar o restar determinados permisos, utilizamos la siguiente sintaxis:
`chmod quien+que archivo`. donde `"quien"` puede ser: `u` (usuario dueño), `g` (grupo), `o` (otros), o bien `a` (todos). Y `"que"` puede ser `r`, `w` o `x`. Los permisos que no se agreguen ni quiten permanecerán como estaban.

Así, si escribimos `chmod u+r archivo` le estamos agregando permiso de lectura al usuario. Si hacemos `chmod o-w archivo` le quitamos permiso de lectura a los demás. Por otro lado, escribiendo la línea `chmod a+x,go-w archivo` le agregamos permiso de ejecución a todos los usuarios, y le quitamos el de escritura a todos los que no son el dueño.

Por otro lado, existe una manera más rápida, pero también un poco más complicada de cambiar los permisos. Esta manera utiliza números octales. Consideramos `r = 4`, `w = 2` y `x = 1`. El valor del permiso que queremos asignar será la suma de los permisos. Y lo tenemos que escribir para el usuario, el grupo y los otros.

Ejemplo: `chmod 644 archivo` asigna permisos de lectura y escritura para el usuario y sólo de lectura para los demás. `chmod 755 archivo` asigna permisos de lectura, escritura y ejecución al usuario y de lectura y ejecución a los demás.

Si deseamos cambiar los permisos dentro de un directorio, de forma *recursiva*, podemos utilizar la opción `-R`.

5.4.2. groups - chgrp

Un usuario puede pertenecer a varios grupos. Para ver a qué grupos pertenece, utilizamos el comando `groups`.

Y el comando `chgrp grupo archivo` nos permite cambiar el grupo de un archivo. Solamente podemos cambiar el grupo a otro grupo que también pertenezcamos.

5.4.3. chown

Este comando solamente lo puede ejecutar el superusuario, y le permite cambiar el dueño de un archivo, pero como no tenemos permiso de superusuario, no podemos utilizarlo.

5.5. Ejercicios

1. Indicar los comandos a utilizar para otorgarle a un archivo cualquiera permisos de:
   lectura y escritura solamente para el dueño del archivo;
   lectura y ejecución para todos los usuarios;
   lectura para todos los usuarios, y escritura solamente para el dueño del archivo.

2. Crear un directorio (por ejemplo **pruebadir**), y algunos archivos dentro (por ejemplo, **arch1 arch2 arch3**). Cambiarle los permisos de lectura, ejecución y escritura al directorio y probar diversos comandos: `echo prueba/*`, `cd prueba`, `touch prueba/arch4`, `rm prueba/arch3`,

`ls prueba`, etc. De este modo se puede apreciar qué significan realmente los permisos de lectura, escritura y ejecución en un directorio.

6. vi

Vi es un editor de texto para consola. Es el editor de texto tradicional de UNIX, y en muchos sistemas es el único disponible, de manera que es importante saber usarlo, aunque más no sea básicamente.

Para comenzar a editar un archivo deberemos escribir: `vi archivo`, o bien ejecutar `vi`, y luego abrir el archivo con el comando adecuado.

En vi existen dos modos de trabajo: un modo de edición y un modo de comandos. Al iniciar el programa, estamos en el modo de comandos. Para ingresar al modo de edición debemos apretar `i`, o bien, **Insert**. Para volver al modo de comandos, utilizamos la tecla **ESC**.

Cuando estemos en el modo de edición, todo lo que ingresemos será texto del archivo. Cuando estemos en el modo comandos, no. A veces lo que escribamos no mostrará ninguna salida inmediata en la pantalla.

6.1. Comandos Básicos

:e archivo	abre el archivo.
:q	sale del programa, solo si ya se grabaron los cambios.
:q!	sale del programa sin grabar los cambios.
:w	graba el archivo.
:w archivo	graba el archivo con ese nombre (eq. Guardar Como)
:wq	graba el archivo y luego sale del programa.

6.2. Teclas de Movimientos

0	inicio de línea.	**$**	fin de línea.
b	anterior palabra.	**w**	próxima palabra.
h	izquierda.	**l**	derecha.
k	arriba.	**j**	abajo.
G	fin de archivo.		

A la mayoría de estos comandos se les puede agregar un número al principio. El efecto de este número será el de multiplicar el efecto del comando por el número ingresado. Por ejemplo, **10j** se mueve 10 lineas hacia abajo.

En el caso de **G**, el número que se le agregue antes puede ser el número de linea al cual deseamos ir. Si deseamos ir a la primera línea del archivo, debemos escribir **1G**.

6.3. Manejo de Texto

Como en cualquier editor de texto, podemos cortar, copiar y pegar.

dd	corta la línea.
dw	corta la próxima palabra.
d$	corta hasta el final de la línea.
p	pega lo que se haya cortado o copiado
u	(undo) deshace la última acción.
yy	copia la línea.
x	corta el caracter.

Muchos de estos comandos también aceptan un número que los preceda, de tal manera que se pueden seleccionar varios caracteres, palabras o líneas a un mismo tiempo.

6.4. Búsquedas

Ingresando **/texto**, (ENTER) nos llevará a la próxima aparición de 'texto'. Para ir a la anterior: **?texto** (ENTER). Una vez que lo hemos buscado, presionando **n** iremos a la siguiente aparición, y presionando **N** iremos a la anterior.

6.5. Otros

CTRL-g	muestra la línea actual y el total de líneas.
o	agrega una línea debajo de la actual, y entra en modo inserción.
a	se coloca en el caracter siguiente al actual, y en modo inserción.

6.6. Más información

Dentro del mismo vi es posible obtener ayuda sobre cómo manejarlo, utilizando el comando `:help`, desde el modo comandos. Para salir de la ayuda y volver al archivo que se estaba editando: `:q`.

Una de las versiones de vi más difundida es el vim (Vi Improved). Tiene muchas más características, como resaltado de sintaxis (syntax highlighting) y muchas otras cosas más. Para los recién iniciados existe un comando `vimtutor`, que nos da unas breves clases sobre el uso de vim.

7. Emacs

A pesar que emacs no se encuentra en todos los sistemas UNIX, y por lo tanto no es una herramienta con la que podemos contar en cualquier caso, es uno de los procesadores de texto más utilizados. En realidad es mucho más que un procesador de texto, puede utilizarse como un lector de correo y de noticias, un navegador web, una interfaz para varios juegos, y hasta un entorno de programación y desarrollo.

Esto es posible porque emacs es principalmente un intérprete del lenguaje LISP, un lenguaje que fue desarrollado para la investigación de inteligencia artificial. Dentro de emacs, es el intérprete de LISP quien ejecuta un editor de texto y el que interpreta cada uno de nuestras entradas. Sin embargo, en esta introducción utilizaremos simplemente sus capacidades como editor de texto.

7.1. Una breve recorrida

Para ejecutar emacs podemos, desde la línea de comandos escribir `emacs`, y dentro del emacs abrir el archivo que queramos editar. O bien, abrir el archivo al entrar al emacs ejecutando `emacs archivo`, es posible, además, comenzar a editar varios archivos a la vez, utilizando `emacs arch1 arch2 arch3`.

Para abrir un archivo dentro de emacs, se utilizan las teclas **Ctrl-x Ctrl-f**. Esto muestra un mensaje en la parte inferior de la pantalla `Find file: ~/`, que indica que a continuación puede ingresarse el nombre del archivo a editar. Al igual que dentro del intérprete de comandos `bash`, la tecla `TAB` nos será de gran utilidad, ya que completará automáticamente el nombre del archivo, o desplegará una lista de los archivos que coincidan con el nombre que se escriba.

Una vez que hemos escrito lo suficiente, se puede grabar utilizando **Ctrl-x Ctrl-s**.

Para salir de emacs usamos las teclas **Ctrl-x Ctrl-c**. emacs preguntará si queremos guardar los cambios hechos, contestando `"y"` se grabarán los cambios, mientras que `"n"` no los grabará, y `"!"` no guardará ningún cambio.

7.2. Meta y Control

Emacs es un editor que no tiene múltiples modos como el vi. Es decir, en cuanto se abre el editor ya está listo para que empieces a escribir y todo lo que se escriba lo incluirá en el archivo que se está editando.

Para acceder a un comando del editor debemos apretar una combinación de teclas. Para eso usamos unas teclas denominadas Meta y Control junto con otras teclas. Estas teclas no se muestran en el texto, son órdenes al editor. La tecla Meta en la mayoría de las PCs modernas no existe y en su reemplazo puede encontrarse la tecla ALT o la tecla 'windows'.

También es posible usar la tecla ESC como reemplazo de la tecla Meta. Pero en ese caso la tecla ESC no es apretada al mismo tiempo que la otra tecla, sino que apretamos primero la tecla ESC, la soltamos y luego apretamos la tecla que dispare el comando o acción del editor.

Por ejemplo, para acceder al tutorial debemos apretar **Ctrl-h t**, esto significa apretar la tecla Control al mismo tiempo que la tecla h y luego la tecla t. Por comodidad, dentro de los tutoriales y ayudas de emacs (y también dentro de esta breve introducción), se escriben estas combinaciones con una C para Control y una M para Meta.

7.3. Ventanas (Buffers)

emacs es un editor que nos permite trabajar con múltiples archivos a la vez. Cada archivo lo deja disponible a través de distintos *buffers*, que son distintas ventanas a las que el usuario puede acceder; puede cambiar de una ventana a otra en cualquier momento o tener varias ventanas de texto a la vista dentro de la misma pantalla (dividiéndola al medio, en cuartos, en octavos, etc).

Para obtener una lista de las distintas ventanas disponibles, podemos apretar **C-x b**, esto dividirá la pantalla actual en dos y en la parte inferior tendremos una lista de todas las ventanas disponibles. Con **C-x o** podemos pasar de la ventana superior a la inferior, y viceversa.

A continuación, algunas otras teclas que pueden llegar a resultar útiles en el manejo de ventanas.

C-x 2	Divide la ventana actual en dos partes horizontales
C-x 3	Divide la ventana actual en dos partes verticales
M-C-v	Avanza una pantalla en la otra ventana
C-x o	Cambia de ventana
C-x b	Muestra una lista de todas las ventanas disponibles
C-x k	Cierra la ventana actual
C-x 0	Cierra la otra ventana

7.4. Algunas combinaciones de teclas

A continuación, algunas otras combinaciones de teclas que pueden resultar útiles en el uso de emacs.

7.4.1. Deshacer y volver a hacer

Para deshacer el último cambio realizado, se utiliza la combinación **C-_**. Esta combinación puede presionarse tantas veces como se desee para ir deshaciendo todos los cambios realizados.

Si, llegado un punto, si quiere volver a realizar los cambios que se han deshecho, simplemente se realiza cualquier cambio (por ejemplo, escribir un caracter) y luego se vuelve a presionar **C-_** tantas veces como sea necesario.

7.4.2. Teclas de movimientos

Dentro de emacs es posible utilizar los cursores para moverse a través del texto del archivo, pero además es posible utilizar estas teclas.

C-n	Baja una línea	**C-p**	Sube una línea
C-f	Avanza un caracter	**C-b**	Retrocede un caracter
M-f	Avanza una palabra	**M-b**	Retrocede una palabra
C-a	Principio de línea	**C-e**	Fin de línea
M-a	Principio de oración	**M-e**	Fin de oración
M-arriba	Principio de párrafo	**M-abajo**	Fin de párrafo
C-v	Sube una pantalla	**M-v**	Baja una pantalla
M-<	Principio de la ventana	**M->**	Fin de la ventana

7.4.3. Teclas para copiar y pegar

En emacs, para poder copiar y pegar, primero se inicia la selección del texto, luego se mueve el cursor hasta el final de la selección (el programa normalmente no indica que se está marcando el texto), y una vez que se llega al final de la selección se presiona la combinación correspondiente a cortar o copiar, según se desee.

C-espacio	Comenzar a marcar
C-w	Cortar la región marcada
M-w	Copiar la región marcada
C-y	Pegar la región que se haya cortado o copiado.

7.4.4. Teclas para realizar búsquedas

C-s	Busca hacia abajo
C-r	Busca hacia arriba
C-s C-s	Repite la búsqueda
M-C-s	Busca utilizando una *expresión regular*
M-%	Reemplaza en el texto, con confirmación

7.4.5. Teclas de borrado

C-d	Borra un caracter
M-d	Borra una palabra
M-del	Borra la palabra anterior
C-k	Borra una línea
M-k	Borra una oración

8. Scripts de Shell - Básicos

Siempre que interactuamos con el shell estamos creando pequeños programas. Cuando a estos pequeños programas los guardamos en un archivo que pueda ejecutarse luego, lo llamamos un "script". Estos scripts sirven, entre otras cosas, para tareas de automatización.

Los scripts de shell son básicamente programas escritos en el lenguaje del shell, se parecen a los archivos .BAT del DOS. Aunque es cierto que, si se van a escribir más de varias decenas de líneas de script, es mejor recurrir a un lenguaje diferente del shell, pensar en que los scripts de shell son solamente pequeños programas sería cometer una injusticia con la flexibilidad y comodidad que ofrecen al usuario.

Normalmente los scripts de shell se crean cuando es necesario combinar en una única acción una tarea rutinaria.

Estos scripts pueden ser muy breves (una sola línea), o bastante largos (varias pantallas). Vamos a empezar con algunos ejemplos muy sencillos.

8.1. Primer Script de Shell

Creamos un archivo que contenga: `who | wc -l`, esto lo podemos hacer utilizando el comando `echo`, con la salida estándar a un archivo, o bien utilizando el editor vi. El archivo lo llamaremos `cuantos`.

Para poder ejecutar este pequeño script, tenemos que utilizar el intérprete del shell. Hay dos formas de hacerlo: `sh cuantos`, invocará al sh para que interprete nuestro script y lo ejecute.

El archivo no lo podemos ejecutar directamente, porque al crearlo, lo hemos creado como archivo de texto, y no tiene permisos de ejecución. Para que podamos ejecutarlo tenemos que agregarle el permiso: `chmod u+x cuantos`. Una vez que le hemos agregado el permiso, nuestro archivo ha pasado a ser un ejecutable, y se lo puede invocar como a cualquier comando. La línea para invocarlo será: `./cuantos`.

8.2. La variable PATH

Llama la atención el **./** que tenemos que agregar al principio para poder ejecutar el archivo. Esto simboliza al directorio actual. Lo que estamos haciendo es diciéndole al shell que busque el archivo "cuantos" en el directorio actual. Esto se debe a que el directorio actual no está en la lista de los directorios en los que el shell busca para encontrar un determinado comando.

Los directorios en los que el shell busca son **/bin**, **/usr/bin** y algunos otros. Están definidos en una variable llamada PATH.

Para poder ver el contenido de la variable PATH, podemos ejecutar desde línea de comandos: `echo $PATH`. Es importante recalcar que la variable se llama **PATH**, pero para ver su contenido le agregamos el símbolo $ adelante, este símbolo es un meta-caracter de shell, como los vistos anteriormente, y le dice al shell que lo que viene a continuación es una variable.

8.3. Variables de entorno

Así como PATH también existen muchas otras variables. Algunas determinadas por el sistema, y otras determinadas por nosotros.

Llamamos *entorno* al conjunto de variables, como el PATH, el nombre de usuario, el directorio actual, el directorio principal, las preferencias de lenguaje, etc. que determinan a la consola que estamos utilizando en este momento. Podemos ver cuales son las variables de nuestro entorno escribiendo: `set`.

A estas variables de entorno, nosotros podemos agregar nuevas variables. Para ello podemos escribir: `variable=valor`. Y haciendo `echo $variable` podremos ver el valor que le asignamos a la variable.

Al ejecutar un programa, este programa recibe una copia de nuestro entorno, donde algunas variables pueden mantenerse (variables exportadas), y otras pueden no estar. Un programa puede modificar las variables que tiene en su entorno, pero no las del entorno anterior. A su vez, dentro de ese programa podemos ejecutar un nuevo programa, que tendrá su propio entorno, con sus propias variables.

Para hacer que los programas hereden las variables que nosotros definamos, existe un comando llamado `export`, que nos asegura que los programas que se ejecuten, reciban esa variable en su entorno.

Ejemplos

- Escriba en la consola `dia=sabado`, cree un script que contenga: `echo Hoy es $dia`. Ejecute el script, a coninuación ejecute `export dia` y vuelva a ejecutar el script. Explique lo sucedido.
- Crear otro script, que contenga: `variable2=valor2;export variable2; echo $variable2`. Ejecutar el script, y luego, desde la línea de comandos `echo $variable2`. Esto no nos mostrará nada, ya que como dijimos antes, no se puede modificar el entorno anterior.
- Crear otro script, que contenga: `cd /; pwd; echo $PWD; export PWD`. Al ejecutar el script, notar que por más que tanto el `pwd` como el `echo` nos haya mostrado que el `cd` había logrado cambiar de directorio, al volver del script, permanecemos en el directorio en el que estábamos al ejecutar el script.

8.4. Parámetros de los comandos

Cuando ejecutamos un comando, puede suceder que necesite parámetros. En el caso de los scripts, los parámetros se encuentran en variables especiales que identificamos como \$1, \$2, \$3, etc. El nombre de la variable nos indica la ubicación del parámetro.

Para ver esto en un script muy sencillo, armaremos un script que contenga la línea `echo $1`, de forma que -al invocarlo- nos mostrará el parámetro que le pasemos al ejecutarlo.

Para tener el caso de un script un poco más útil, vamos a hacer un script que nos permitirá convertir rápidamente en ejecutables a los próximos scripts que realicemos: `chmod +x $1`. Le pondremos de nombre **cx**. Para hacer ejecutable este script, vamos a aprovechar lo que ya escribimos y haremos: `sh cx cx`.

Podemos mejorar este script, de tal manera que reciba más parámetros y pueda hacer ejecutables más archivos a la vez. Podríamos escribir, por ejemplo. `chmod +x $1 $2 $3 ...`, pero existe una manera más elegante (y más eficiente): `chmod +x $*`. Al encontrar el \$*, el shell lo reemplazará por todos los parámetros que haya recibido el script.

Por último, veremos la utilidad de las comillas dobles ("), en su relación con los parámetros. Usaremos un script que contenga la línea: `echo "Buen día $1"`. Al ejecutarlo con un parámetro (por ejemplo, Pedro), la salida del comando será `Buen día Pedro`.

Si en el ejemplo anterior hubiéramos utilizado comillas simples, la salida del comando hubiera sido `Buen día $1`, ya que dentro de las comillas simples el meta-caracter `$` no es interpretado, mientras que dentro de las comillas dobles si.

9. Filtros - Básicos

Existen una gran variedad de programas que reciben una entrada, la procesan y emiten una salida, a este tipo de programas le llamamos filtros. Estos son una parte fundamental del sistema ya que relacionando filtros simples se puede conseguir prácticamente cualquier cosa, a nivel de administración de archivos y scripts..

Entre estos filtros están:

cat	para mostrar la entrada.
grep, tail, head y cut	para seleccionar una parte de la entrada.
less y more	para paginar la entrada.
sort	para ordenar la entrada
tr	para alterar la entrada en forma ordenada.
comm y diff	para comparar dos entradas.
wc	para contar la entrada.

9.1. grep

Un comando muy útil que nos proporciona UNIX es `grep`. Su función principal es imprimir por pantalla solamente las líneas que concuerden con un patrón que nosotros le indicamos.

Por ejemplo: `ls -l | grep archivo`, nos mostrará todas las líneas del `ls -l` que concuerden con `archivo`.

Algunas opciones de grep son:

- `-i` (ignore case) que no tiene en cuenta mayúsculas y minúsculas.
- `-n` para que muestre el número de línea dentro del archivo.
- `-v` para que muestre las líneas que no concuerdan con la palabra ingresada.

Vamos a hacer un ejemplo de una agenda sencilla, utilizando el comando grep y unos scripts de shell. En primer lugar necesitamos tener un archivo con unas cuantas líneas de la forma: “nombre: teléfono”. Por lo menos cuatro o cinco líneas, como para probar este ejemplo.

Una vez que hemos creado el archivo, crearemos el script, que se llamará 110, y deberá contener la siguiente línea: `grep -i "$*" agenda`. Y para buscar un teléfono dentro de nuestra agenda haremos: `110 nombre`.

9.2. tail

El comando `tail` nos permite ver la cola de un archivo, es decir sus últimas líneas (últimas 10 por omisión). Esto, aunque a simple vista no parezca útil, resulta de gran utilidad para ver archivos grandes como, por ejemplo, logs (archivos donde se guardan mensajes de estado y/o de errores).

Para probar el comando vamos a utilizar los logs del sistema, que se suelen guardar en /var/log, en particular /var/log/messages es el archivo que contiene gran parte de los mensajes del sistema. Podemos hacer `tail /var/log/messages` y compararlo con la salida de `cat /var/log/messages`.

Además a `tail` podemos indicarle cuántas líneas queremos que nos muestre. Con la opción -20, por ejemplo, le diríamos que nos muestre 20 líneas y así con cualquier número que le pasemos. También podemos decirle que nos muestre el contenido del archivo a partir de una determinada línea, por ejemplo `tail +5 archivo` nos mostrará el contenido del archivo desde la línea 5 en adelante.

Otro comando similar a `tail` es `head`. Su comportamiento es muy similar, pero en lugar de mostrar las últimas líneas de la entrada, muestra las primeras.

9.3. sort

Este comando nos permite obtener una salida ordenada, si hacemos `ls | sort` obtendremos la salida de ls con un orden alfabético.

Ahora probemos `ls -i | sort`, `sort` ordena la salida de `ls` alfabéticamente, eso quiere decir que los números no van a estar ordenados de menor a mayor sino en orden alfabético. Para ordenarlos numéricamente existe la opción -n, `ls -i | sort -n` producirá una salida ordenada de menor a mayor. Si quisiéramos, en cambio, que lo ordene de mayor a menor, debemos usar `ls -i | sort -nr`, es decir -r ordena en orden inverso.

También podemos ordenar por una determinada columna, para entender esto de una manera más clara veamos la salida de `ls -l`, estas líneas pueden ser ordenadas por tamaño, por nombre, por fecha, etc. Para eso tenemos que indicarle cuál columna tiene que ordenar y con qué criterio. Notar que sort cuenta las columnas empezando de 0.

Por ejemplo:

`ls -l | sort +4nr` ordena la salida de `ls` por el tamaño dejando los más chicos al final. En este caso, la n, que indica ordenamiento numérico es opcional, ya que `sort` decide automáticamente el ordenamiento.

`ls -l | sort +4n +8` ordena la salida de `ls` primero por tamaño y luego por el nombre del archivo. En este caso, es necesario poner la n, que indica ordenamiento numérico.

También existe un opción -u que nos permite eliminar las lineas iguales adyacentes de la salida de sort. Es decir que si nuestra salida produce dos líneas iguales una debajo de otra, veríamos solamente una línea.

9.4. tr

Este comando nos permite transliterar caracteres, es decir cambiar determinado caracter por otro. Si escribimos `tr 'a-z' 'A-Z' < archivo`, veremos que tr transforma todas las minúsculas en mayúsculas.

Los parámetros del ejemplo anterior son rangos (de la 'a' a la 'z' y de la 'A' a la 'Z'), pero no es necesario que lo sean. Lo que sí nos exige `tr` es que los parámetros tengan la misma cantidad de caracteres, o bien que el segundo sea un solo caracter. En este último caso reemplazará todos los caracteres del primer parámetro con el del segundo.

La opción `-c` indica que deben transliterarse todos los caracteres que no estén incluidos en el primer parámetro.

Ejemplo difícil:

`cat archivo | tr -c 'a-zA-Z' '\n' | tr 'A-Z' 'a-z' | sort -u`

El primer `tr` convierte todo lo que no sea letras en `\n` (retorno de carro), el segundo convierte todas las mayúsculas en minúsculas y el `sort` del final las ordena alfabéticamente eliminando las líneas iguales, con lo que obtenemos una palabra por línea.

9.5. comm

Este comando nos sirve para comparar el contenido de dos archivos, línea por línea. Para poder probar este comando, debemos crear dos archivos parecidos, que tengan unas cuantas líneas en común, y otras distintas. Luego, debemos hacer: `comm archivo1 archivo2`

La salida está separada en tres columnas: en la primera columna estarán las líneas que están en el primer archivo y no en el segundo, en la segunda columna veremos las líneas del segundo archivo que no estén en el primero y por último, en la tercera columna, las que están en los dos.

Además podemos indicar cuál de esas columnas no queremos que muestre, por ejemplo, utilizando: `comm -12 arch1 arch2` muestra solamente la tercera columna, es decir, las líneas que están en ambos archivos.

El problema que tiene `comm` es que espera que las líneas estén ordenadas alfabéticamente, o en la misma posición dentro del archivo. De modo que puede suceder que archivos muy similares sean procesados por `comm` como archivos muy diferentes. Para procesamiento más avanzado de diferencias de archivos puede utilizarse el comando `diff`.

Ejemplo:

En **/usr/share/dict/** normalmente tenemos un archivo diccionario, en el cual encontramos una palabra por renglón. En especial, el archivo **words** suele ser un symlink al diccionario del idioma de nuestro sistema.

Lo que vamos a hacer es armar un pequeño analizador ortográfico, usando los filtros recién vistos. Utilizaremos `comm` para verificar si la palabra está en el diccionario, y que nos avise si no está. Primero deberemos llevar el texto a una palabra por línea ordenada alfabéticamente (que es el ejemplo que utilizamos con `tr`). Y luego, utilizar `comm` para comparar con el diccionario.

El comando completo será:

```
cat archivo | tr -c 'a-zA-Z' '\n' | tr 'A-Z' 'a-z' | sort -u |
comm -23 - /usr/share/dict/words
```

Notar que el - como parámetro del `comm` le indica que use la entrada estándar para leer el primer archivo.

10. Filtros - Avanzados

Existen, además de los filtros que ya vimos, otros dos filtros muy poderosos, llamados `sed` y `awk`. Tienen una sintaxis tan avanzada que se los considera pequeños lenguajes de programación. En este curso no los vamos a estudiar.

Sin embargo, gran parte de la potencia de estos es su manejo de expresiones regulares que es el mismo que el de `grep`, que veremos a continuación.

10.1. grep

Como ya vimos, podemos hacer `grep patrón archivos` y veremos las líneas que concuerdan con el patrón en esos archivos,

La gran fortaleza de `grep` se encuentra en la sintaxis del patrón. Hasta ahora, el patrón que ingresábamos era simplemente una palabra, y eso le indicaba a `grep` que debía encontrar las líneas que contuvieran esa palabra.

Sin embargo, en ese patrón podemos incluir lo que se llama una *expresión regular*. Las expresiones regulares son cadenas de caracteres que permiten transmitirle a los filtros una búsqueda mucho más avanzada. Tenemos la posibilidad, por ejemplo, de indicar si queremos que algo esté al principio o al final de un archivo, si queremos que esté repetido una determinada cantidad de veces, etc.

- El símbolo ^ indica comienzo de línea, es decir que el patrón que esté a continuación debe estar al principio de la línea.

Por ejemplo, podemos querer ver todas las líneas del comando `who` que comienzan con un nombre de usuario en particular. Esto lo haríamos ejecutando: `who | grep ^usuario`.

- De una manera similar $ al final de un patrón significa que debe estar ubicado al final de la línea.
- El . actúa como el ? en el shell, concuerda con cualquier caracter, una sola vez.

  Por ejemplo, si hacemos `ls -l | grep '^.......rw'` veremos los archivos que tengan permisos de lectura y escritura para el resto mundo.
- También tenemos un manejo de rangos `[...]` muy parecido al del shell.
- El * nos permite que el último caracter se encuentre 0 o más veces. Es decir, utilizando `.*` obtendremos el mismo comportamiento que el del `*` de shell.

  Por ejemplo, `ls -l | grep '^.*rw'` puede mostrarnos los archivos que tengan algún permiso de escritura y lectura, o bien algún archivo que contenga las letras rw en su nombre.

Un problema común es que muchos de los caracteres que `grep` usa como caracteres especiales son los mismos que usa el shell como meta-caracteres y esto puede llegar a confundirnos mucho, por lo que usamos los patrones entre comillas simples ('...') para evitar problemas de interpretación del shell. Pero también podemos querer que el grep no interprete determinado caracter en ese caso usamos `\` .

10.2. egrep o grep -E y fgrep o grep -F

En realidad, grep es el primero de una familia de comandos, egrep tiene un set de expresiones regulares más completo, mientras fgrep esta optimizado para manejo de archivos grandes, pero solo busca literales.

En la actualidad la versión GNU de estos comandos resume los tres dentro de grep con las opciones -E y -F, pero existen más, como el ngrep (network grep), grepmail, etc.

10.3. Expresiones de grep, egrep

Se listan a continuación las expresiones de grep y egrep, por orden de precedencia, con una explicación somera de su significado.

Expresión	Significado
c	cualquier caracter no especial, concuerda a si mismo
c	cancela significado especial de c
^	inicio de línea
$	fin de línea
.	cualquier caracter individual
`[...]`	cualquiera de los caracteres en ...; incluye rangos de tipo a-z
`[^...]`	cualquiera de los caracteres que no esté en ...; también se incluyen los rangos a-z
r*	cero o más ocurrencias de r
r+	una o más ocurrencias de r (egrep)
r?	cero o una ocurrencia de r (egrep)
`r1|r2`	expresión r1 o expresión r2 (egrep)
\\(**r**\\)	agrupar expresión regular r (grep)
(r)	agrupar expresión regular r (egrep)
num	lo que concordó con la num-ésima expresión regular agrupada

Ante la pregunta posible, que puede surgir mirando esta tabla, de cuál es el significado de `(r)` cuando se utiliza grep, la respuesta es que simplemente concuerda con los paréntesis, como cualquier caracter sin significado especial

10.4. Ejercicios

Utilizando el diccionario que se encuentra en el directorio **/usr/share/dict**, encontrar las expresiones regulares que concuerdan con los siguientes tipos de palabras.

1. Palíndromos de 3 letras (ej: ala, asa, ata).
2. Palíndromos de 4 letras (ej: erre).
3. Palíndromos desde 3 a 9 letras (ej: salas, rallar, anilina)
4. Palabras que tienen todas sus letras ordenadas alfabéticamente.
5. Palabras que tienen todas sus vocales ordenadas alfabéticamente.
6. Palabras que tienen todas las vocales. (ayuda: utilizar varios `grep` encadenados).

11. Más scripts de Shell

Además de las herramientas para manejo de variables que se explicaron anteriormente, el shell nos permite utilizar herramientas para manejo de ciclos y para estructuras condicionales, veremos a continuación cómo utilizar y manipular los parámetros y luego ejemplos que utilizan **if**, **for**, **while** y **case**, junto con varias herramientas y características del shell, como manejo de funciones, valores de retorno, etc.

11.1. Parámetros

Como ya hemos visto, los scripts de shell pueden recibir y manipular parámetros. Estos parámetros se representan dentro de los scripts como $1, $2, etc. El conjunto de todos los parámetros está representado por $*, mientras que la cantidad de parámetros está representada por $#.

Existe, además un comando llamado **shift** que permite eliminar el primer parámetro de la lista, y correr todos los parámetros. De tal manera que el segundo parámetro pasa a ser $1 y así.

Veremos un ejemplo sencillo:

```
echo "Cantidad de parámetros: $#"
echo "Primer parámetro: $1"
shift
echo "Segundo parámetro $1"
shift
echo "El resto de los parámetros $*"
```

El comando **set** nos permite ver todas las variables de entorno. Pero además, nos permite asignarle valor a los parámetros $1, $2, etc. Por ejemplo:

```
set Viva GNU Linux
echo "Primer parámetro: $1"
echo "Segundo parámetro: $2"
echo "Tercer parámetro: $3"
```

11.2. if

Es la estructura que permite ejecutar los comandos solamente si se cumple una determinada condición. La sintaxis más usual:

`if [ condición ]; then`
`  comandos`
`else`
`  comandos`
`fi`
	`if [ condición ]; then`
	`  comandos`
	`fi`

En realidad, los corchetes `[ ]` son un comando en si mismo, también llamado `test`, por eso, para armar la condición utilizamos la sintaxis de test. (para más información: `man test`).

Las condiciones serán de este estilo:

!condición	Si condición es falsa
condición1 -a condición2	Las dos condiciones son verdaderas
condición1 -o condición2	Una de las dos condiciones es verdadera
cadena-de-caracteres	La cadena no esta vacía
-z cadena-de-caracteres	La cadena esta vacía
cadena = cadena	Las dos cadenas son iguales
cadena != cadena	Las cadenas son distintas
entero -eq entero	Los enteros son iguales

Por otro lado, también podemos colocar otra cosa que no sea el comando test, otros programas, que le devuelvan al if un cero o un uno, que el if luego interpretará como *false* o *true* respectivamente. O también, la palabras **true** y **false**.

Realizaremos un script que escriba un mensaje si hay más de un usuario conectado y otro diferente si hay sólo un usuario conectado.

```
cuantos=`who | wc -l`

if [ $cuantos -gt 1 ]; then
        echo "Hay $cuantos usuarios conectados"
else
        echo "Estás solo en el mundo"
fi
```

En este script, primero guardamos la cantidad de usuarios conectados al sistema en la variable **cuantos**. Esto lo hacemos utilizando las comillas invertidas, que son un recurso que nos provee el shell para utilizar la salida de los comandos en nuestro código; cada bloque encerrado por las comillas invertidas es reemplazado por la salida del comando en cuestión. Otra forma de hacer esto es utilizando una sintaxis alternativa a las comillas invertidas `$(...)`, esta forma tiene la ventaja que podemos anidarla fácilmente.

Luego utilizamos la estructura de **if** para compararlo con 1, la opción -gt del comando **test** significa mayor que.

Recordar: el comando [normalmente es sinónimo del comando test.

11.3. for

Es una estructura que permite una iteración durante un número determinado de veces.

La sintaxis:

```
for variable in lista; do
  comandos
done
```

La variable, cuando se la coloca al lado del for, no lleva el signo $ adelante, pero si en los comandos se hace referencia a ella, se debe escribir $variable.

La *lista* puede ser:

Una lista de números. Ej: 1 2 3 4

Una lista de archivos. Ej: *.java

Una lista de argumentos. Ej: $*

Por ejemplo, si quisieramos hacer copias de varios archivos agregandoles una extensión, podriamos hacer.

```
for archivo in *; do
        cp "$archivo" "$archivo.bak"
done
```

A continuación realizaremos un script que recibe por parámetro todos los usuarios que queremos saludar y les manda un saludo a cada uno solamente si están conectados al sistema.

```
for i in $*; do
        if who | grep "^$i" > /dev/null; then
                write $i << EoT
Hola $i!
EoT
        fi
done
```

Esta vez utilizamos la estructura **for** para procesar los argumentos de nuestro comando. Para cada uno de los parámetros recibidos, verificamos si ese usuario está conectado o no. Para esto, abusamos un poco del if y demostramos que no es obligatorio usar test con el if. Aquí utilizamos la salida de un comando.

Si el usuario está conectado usamos el comando **write** (que le manda un mensaje a la consola del usuario), para enviar un saludo a cada uno.

Para poder mandar el mensaje, utilizamos un meta-caracter de shell muy especial. Este meta-caracter comienza con `<< EoT`, y con esto le indica al shell que la entrada de este comando es todo lo que ingresemos a continuación hasta la que encuentre una línea que solamente contenga la cadena `EoT`.

Esta cadena podría ser cualquier otra, podríamos por ejemplo poner `<< blablabla`, y luego deberíamos terminar el texto con una línea que contenga `blablabla`.

11.4. while

Es una estructura que permite una iteración hasta que una determinada condición no se cumpla.

La sintaxis:

```
while [  condición ]; do
  comandos
done
```

La condición es equivalente a la del if.

A continuación veremos otra manera de procesar los parámetros recibidos.

```
while [ "$*" ]; do
        echo $*
        shift
done
```

Este simple script imprime en cada línea cada uno de los parámetros recibidos. Vemos que si omitimos las `"  "` el comando test nos devuelve un error, pues espera un solo parámetro.

11.5. case

En casi todos los lenguajes de programación encontramos una estructura que nos permite realizar distintas acciones según el valor de una determinada variable.

En el caso del shell esta estructura es **case**. Y se utiliza con la siguiente sintaxis:

```
case $variable in
            patron1)
                    comandos
                    ;;
            patron2)
                    comandos
                    ;;
esac
```

El patrón con el que comprobamos al utilizar **case** es similar al que utiliza el shell (* para cualquier cadena, ? para cualquier caracter, [ ] para rangos, \ para escapar valores especiales y ' ' para escapar cadenas completas), además de un muy cómodo `|` para usarlo como o.

Para ver cómo es su funcionamiento, estudiaremos un ejemplo, en el que utilizamos el comando **cal**. Este comando nos permite visualizar calendarios. Utilizado sin ninguna opción nos muestra el mes actual. Al agregar un número a continuación (por ejemplo **cal 2020**) nos muestra el calendario de ese año. Y si le pasamos dos parámetros (por ejemplo **cal 11 2002**) nos muestra el calendario de ese mes y ese año.

Este comando no entiende nombres de meses (enero, febrero, etc), y al ejecutar **cal 10** la salida será el calendario del año 10, no del mes 10 del año actual, que es más probablemente nuestra expectativa. El script que realizaremos permitirá obtener un nuevo comando (ncal) que tenga estas funcionalidades.

```
# Analizamos los parámetros.
case $# in
   0) set `date`    # Si no vino ningún parámetro,
```

```
      m=$2          # mostramos el calendario del mes actual
      y=$6          # y del año actual.
   ;;
   1) m=$1          # Si vino un sólo parámetro,
      set ‘date‘    # mostramos el calendario de ese mes
      y=$6          # y del año actual.
   ;;
   2) m=$1          # Si vinieron dos parámetros,
      y=$2          # mostramos el calendario de ese mes y año.
   ;;
esac

# Seleccionamos el mes que corresponde.
case $m in
   jan*|Jan*|ene*|Ene*) m=1 ;;
   feb*|Feb*)           m=2 ;;
   mar*|Mar*)           m=3 ;;
   apr*|Apr*|abr*|Abr*) m=4 ;;
   may*|May*)           m=5 ;;
   jun*|Jun*)           m=6 ;;
   jul*|Jul*)           m=7 ;;
   aug*|Aug*|ago*|Ago*) m=8 ;;
   sep*|Sep*)           m=9 ;;
   oct*|Oct*)           m=10 ;;
   nov*|Nov*)           m=11 ;;
   dec*|Dec*|dic*|Dic*) m=12 ;;
   [1-9]|10|11|12)      ;;           # El mes era numérico.
   *)                   y=$m; m=""; # Sólo vino el año.
esac

# Llamamos al calendario con el mes que elegimos.
cal $m $y
```

Un uso muy frecuente del case de shell se encuentra en los *demonios* de GNU/Linux. Los demonios son los servicios que se ejecutan en el sistema más allá de qué usuario es el que está loggeado. Suelen existir scripts de shell que se encargan de iniciarlos, detenerlos, reinciarlos, etc.

11.6. Funciones

El shell nos provee de un manejo de funciones. Una función es un conjunto de instrucciones encerradas en un bloque de código, una caja negra que hace determinado trabajo. Deben ser declaradas antes de poder llamarlas, y para llamarlas alcanza con tipear su nombre.

```
# declaración de función
función () {
        comandos
}
# declaracion alternativa.
function funcion {
        comandos
}

función # Llamado a función
```

Las funciones reciben sus parámetros en \$1, \$2, \$*, etc. Y devuelven valores entre 0 y 255, esto se consigue con la instrucción de shell **return**. El valor que devuelven puede ser accedido en el script en la variable \$?. Normalmente, las funciones acceden a variables globales, para poder usar locales se debe usar la instrucción de shell **local**.

A continuación, algunos ejemplos que se proponen armar una calculadora en shell.

```
res=0; rem=0
add () {
        local i
        case $# in
        0)      res=0 ;;
        *)      res=$1
                shift
                for i in $*; do
                        let res+=$i
                done
                ;;
        esac
}
```

Se trata de una función que suma todos los parámetros que recibe. Si no recibe ningún parámetro el resultado es cero. En cambio, si recibe algún parámetro, inicializa la variable *res* con el valor del primer parámetro, que luego borra de la lista de parámetros con el comando **shift**. Y a continuación suma uno a uno, todos los otros parámetros, utilizando el comando de shell **let**, que permite realizar operaciones aritméticas simples.

Así como la suma, podemos ver también la multiplicación y la división.

```
mul () {
        local i
        case $# in
        0)
                res=0
                ;;
        *)
                res=$1
                shift
                for i in $*; do
                        let res*=$i
                done
                ;;
        esac
}
div () {
        local i
        case $# in
        2)
                case $2 in
                0)
                        echo "Error: división por cero";
                        res=0; rem=0
                        return 1
                ;;
```

```
                *)
                        let res=$1/$2
                        let rem=$1%$2
                ;;
                esac
        ;;
        *)
                res=0; rem=0
        ;;
        esac
}

add 3 255 123 123 30 29 4
mul $res 2 2
div $res 0
echo $res; echo $rem
```

Existen aún más capacidades para los scripts de shell, que no se explican en este curso. Más información puede encontrarse en la gran documentación disponible en Internet, o leyendo el manual de bash.