Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

Departamento de Electrónica

Asignatura: Informática I

Primeros pasos con un Sistema Operativo

Autor: Ing. Alejandro Furfaro. Profesor Titular

Índice de contenido

Sistema Operativo.

Cualquier curso de programación suele comenzar con los detalles iniciales de un lenguaje determinado, pero rara vez se detiene a describir el entorno de desarrollo que se requiere, y sobre todas las cosas dar algunos rudimentos básicos del Sistema Operativo sobre el que se van a desarrollar y ejecutar las aplicaciones que componen las prácticas del curso.

En nuestros cursos de Informática I y II así como en las asignaturas Técnicas Digitales posteriores se trabaja sobre un Sistema Operativo conocido como Linux.

¿Porque Linux?. Porque es de licencia libre, y por lo tanto es ampliamente difundido en el mundo académico. Es sumamente estable y posee, además de la disponibilidad absoluta de su código fuente, de abundante documentación para su uso y de sus "internals", hecho que sumado a los diferentes foros de soporte de las diferentes distribuciones hace que actualmente sea tan sencillo o tan complejo de instalar como otros Sistemas Operativos comerciales bajo licencia, pero no tiene secretos ni partes oscuras. Todo está ahí, al alcance de un editor. Basta finalmente con ver el programa fuente para salir de dudas.

Antes de comenzar.. ¿Que es un Sistema Operativo?

Buena pregunta para quienes nunca han ensayado una manera formal de definir un Sistema Operativo. Probablemente la mayoría de los lectores familiarizados en principio con el popular sistema Windows, piensen para sí: "Un sistema Operativo es .... como Windows".

Windows, efectivamente es un Sistema Operativo. Pero lo anterior es un ejemplo y no una definición.

Un sistema operativo es una colección de programas que se encargan de administrar los recursos del computador, proveyendo a los diferentes usuarios que pueden estar logueados al sistema la posibilidad de utilizar los recursos de hardware y software del mismo sin necesidad de conocer detalles escabrosos y de manera segura para sí y para el resto. Este hecho es muy interesante ya que es muy común ver lo que ocurre cuando se trabaja con esquemas de protección endebles entre las diferentes aplicaciones que se ejecutan en un computador.

El sistema operativo se instala en la memoria del computador cuando este se enciende y toma el control permitiéndonos ingresar al sistema (login), utilizar aplicaciones y a través de éstas acceder a los recursos de hardware del sistema, proveyendo además una colección de funciones a las que se invoca mediante llamadas predefinidas por el sistema y que devuelven resultados de manera también definida. Estas especificaciones se conocen como *API* (Aplication Programming Interface) y son especificaciones de funciones en lenguaje C. La implementación de las API dentro del Sistema Operativo, es

decir, la forma en que se resuelve la llamada efectuada desde un programa de aplicación, se conocen como *System Calls*.

El conjunto de programas que constituyen el Sistema Operativo propiamente dicho y la implementación de las *System Calls* se denomina *kernel* (núcleo).

Por lo general el sistema operativo posee una interfaz de usuario por lo menos denominada *Shell*. El *Shell* es un intérprete de comandos que se encarga de traducir los pedidos del usuario e invocando las S*ystem Calls* apropiadas los lleva adelante.

Al inicio de los sistemas de cómputo la interfaz del *shell* era en modo texto, y aun hoy se sigue utilizando debido a su agilidad y a la potencia de los comandos en muchos casos. Esta interfaz se conoce actualmente como modo consola.

Con el devenir de los progresos en materia de Hardware el *shell* pasó también a versiones GUI (Graphic User Interface). Este es el modo nativo del Sistema Operativo Windows, aunque los primeros antecedentes de GUI's se registran en los sistemas X de Unix, y en las Mac.

En Windows no está muy clara la división entre el *kernel* y el *shell*, ya que el sistema no permite cambiar el escritorio nativo de Windows por otro cualquiera (de hecho no existen otras alternativas).

En los sistemas Unix-like y en particular en Linux, es muy común poder instalar diferentes escritorios y arrancar con el que mas nos guste. Ejemplos mas comunes: KDE, Gnome. Incluso se puede configurar el sistema para que arranque con un *shell* de consola de texto y trabajar solo en ese modo, aun si se ha instalado algún escritorio gráfico, el cual podemos optar por no utilizar durante nuestra sesión de trabajo, o si se lo quiere emplear se lo levanta mediante el script *startx*. No obstante para adaptarse a los requerimientos de la gran mayoría de los usuarios los sistemas Linux se configuran automáticamente para poder arrancar con un escritorio gráfico de modo que los usuarios que lo exploran y que obviamente suelen utilizar Windows, se encuentren mas familiarizados desde el inicio.

Primer cuestión a destacar: el *shell* es solo una aplicación encargada de la interacción con el usuario. ***No es parte del sistema operativo***. De hecho en el mundo Unix (y por ende Linux) cuando se menciona el kernel se habla del Sistema Operativo, y viceversa.

Para acceder al Hardware, el sistema operativo posee una interfaz de muy bajo nivel llamada Device Drivers (Manejadores de Dispositivos). Estos componentes de software, son parte del kernel y su misión es acceder al hardware de sistema en forma directa. Las aplicaciones NO PUEDEN efectuar este acceso en los sistemas operativos modernos ya que se trata de entornos multiusuario en los que la estabilidad del sistema y de las aplicaciones que se ejecutan dentro de él debe estar garantizada (aunque a veces no nos lo parezca ;))

En líneas generales, la organización de estos módulos es como se ilustra en la Fig. 1.

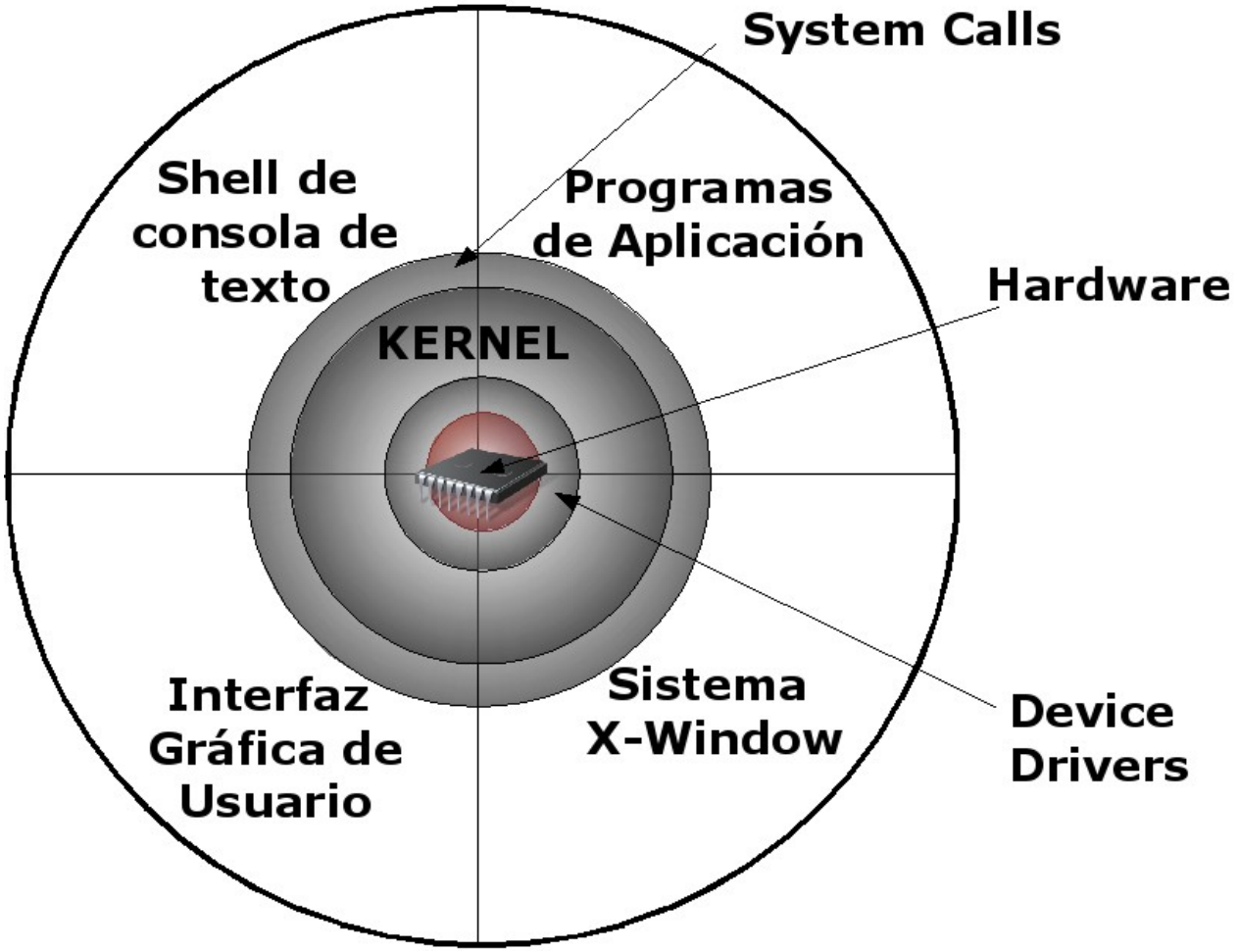

Fig.1 Organización de un Sistema Operativo.

Que es Linux?

No todo el mundo sabe que Linux es una implementación de un kernel basado en UNIX. Por lo tanto los entornos de operación y desarrollo de aplicaciones son muy similares. Tal es así que cualquier programa escrito en UNIX puede compilarse y ejecutarse en Linux. Además algunas versiones de programas comerciales escritos para UNIX pueden instalarse directamente y ejecutar sobre Linux sin inconvenientes (notar que hablamos compatibilidad a nivel binario).

Linux fue desarrollado originalmente por Linus Torvalds en la Universidad de Helsinki, como trabajo de Tesis final. Luego de su presentación y graduación Linus lanzó una invitación a la comunidad de internet para contribuir al desarrollo de un sistema operativo basado en UNIX pero bajo la licencia GPL (General Public Licence) basada en el concepto GNU (sigla recursiva que

significa GNU's Not Unix) impulsado por Richard Stallman. Así creció hasta transformarse en lo que es actualmente: Un Sistema Operativo completo, estable y sobre todas las cosas de fuentes abiertos. No utiliza código de ninguna implementación comercial de UNIX.

Distribuciones

Linus Torvalds se encarga junto con un grupo de desarrolladores de desarrollar y mantener el kernel. Nada mas. Las diversas y por cierto muy numerosas aplicaciones que se conocen para Linux son producto de otros desarrolladores que adhieren a la licencia General Pública (GPL).

Cualquier persona que quiera hacer una contribución al desarrollo del kernel lo puede hacer pero el filtro final y control de consistencia con el resto del kernel queda a cargo de Torvalds y su equipo. Así funciona.

Linux se entrega al público usuario mediante distribuciones. Claro que, un usuario sumamente experto, puede tomar los fuentes de la versión de kernel que desee, en http://www.kernel.org, descargarlos en su computador, bajar el compilador gcc, de http://gcc.gnu.org, compilar el kernel, y descargarle las aplicaciones que desee, pudiendo tener que compilar una por una o en algunos casos encontrar ya disponible un paquete en los formatos mas comunes: rpm, o deb, por citar los mas populares.

Afortunadamente para los humildes mortales existen las distribuciones. ¿que son?. Diversas comunidades de desarrolladores se dedican a compilar la última versión del kernel, agregarle los shells (texto y gráfico), aplicaciones y distribuirlo en forma de un sistema operativo instalable desde CD's o DVD, igual que Windows por ejemplo.

Diversas empresas/organizaciones se han abocado a este curioso modelo de negocio que consiste en distribuir software por el que no se cobra en concepto de licencias de uso. ¿Como resuelven sus ingresos entonces?. Una de las fuentes de ingreso de estas compañías es el cobro de servicios profesionales a empresas que requieren soporte técnico, otras fuentes son consultoría, pero nunca se cobra por el software en sí.

Al grano. Cada empresa/organización le pone un nombre a la distribución, le desarrolla un instalador customizado que podrá ser mas o menos amigable, crea las imágenes de CD's o DVD, las pone en internet, y solo hay que bajarlas, quemar los discos, y a instalar.

Así nos encontramos con diferentes “nombres” de Linux: Debian, Ubuntu, Red Hat, Slackware, Mandriva, Susse, Knoppix, por citar solo los mas conocidos. Hay muchos mas. Lo que debe tenerse siempre presente es que Linux es Linux. El kernel. Y este no varía independientemente de la distribución (o distro como suelen decirle también) que adoptemos.

Linux

El Shell

Ya vimos cual es su función, y la posibilidad de disponer de varios shells simultáneamente en un mismo sistema operativo.

Para comenzar a caminar en Linux es necesario conocer los rudimentos del shell a pesar de que en la vida real todos los usuarios (incluido este autor) inician su sistema con una interfaz gráfica. Pero lo cierto es que es mas que necesario tener sesiones de shell activas a punto tal que ciertos comandos son indispensables para la operación del sistema y se ejecutan desde una consola.

La Fig. 2 da fe de ello

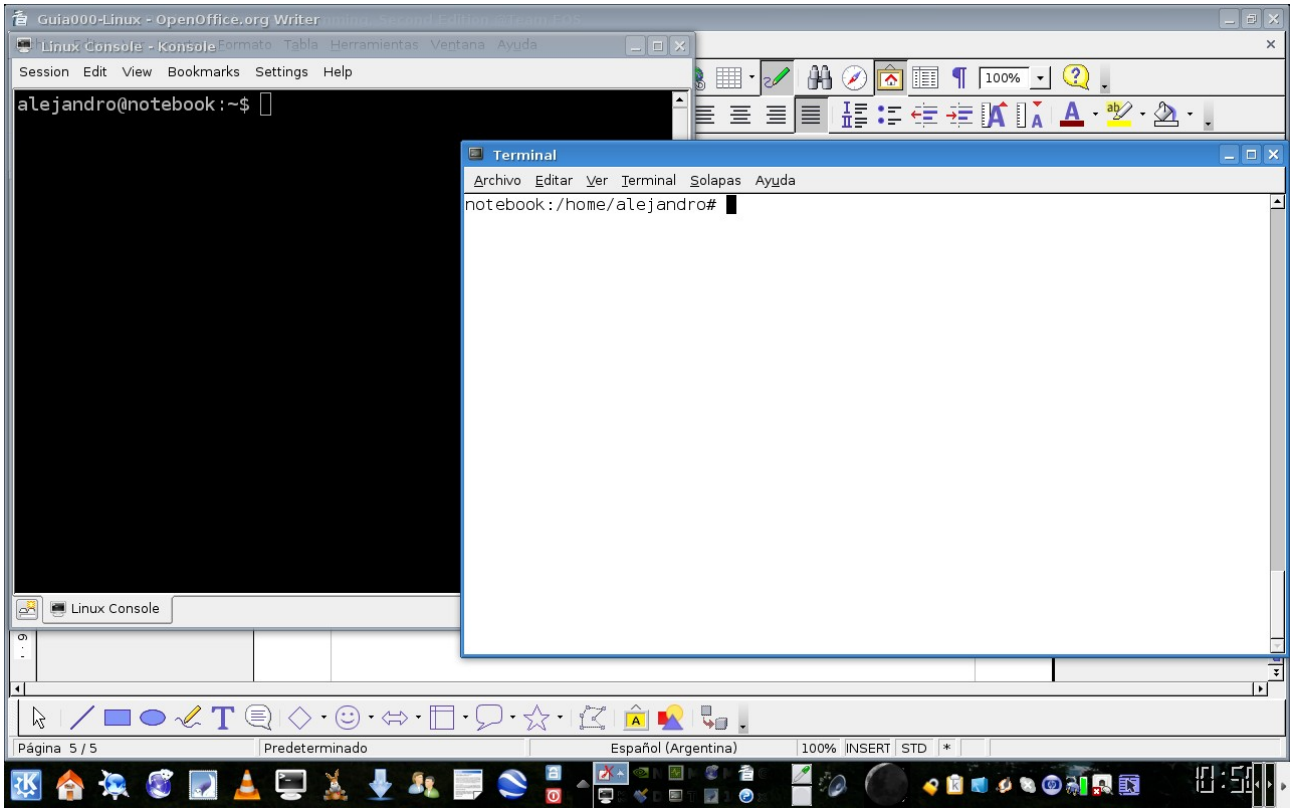

Fig. 2. Un entorno gráfico con dos sesiones de consola en modo gráfico para uso de shell

En primer lugar debemos saber que el *shell* no solo es un intérprete de comandos sino una muy potente herramienta de scripting. Cada vez que tipeamos un comando el *shell* lo asume como un ejecutable binario o como un *script*. Un ejecutable binario es el resultado de la edición y compilación y linkeo de un programa determinado. Un *script* es un archivo de texto que contiene comandos binarios u otros *scripts* para ejecutar, y que además puede incluir sentencias de control de flujo. Muy poderoso. El lector estará preguntándose como es posible que el *shell* ejecute un comando que es un mero archivo de texto. Antes de que comience a hacer comparaciones con otros sistemas, que

lo llevarán a conclusiones erróneas, es necesario remarcar que los archivos en todos los sistemas Unix poseen atributos y permisos, que regulan su acceso para determinados usuarios que satisfagan dichos permisos, y para delimitar que es lo que se puede o no se puede hacer con ellos. Por ello, un archivo de texto con permiso de ejecución es ejecutable por el *shell*.

Y ahora???

En este punto debemos advertir que el *shell* es un entorno que en principio resulta bastante árido. Si no se conocen algunos comandos básicos es muy difícil comenzar a trabajar.

Y además los comandos son en general sumamente potentes y poseen por lo tanto una cantidad de opciones que resulta muy ingrato retener en la memoria, especialmente para estudiantes de Ingeniería que deberán educar su mente para la deducción racional y no para la memorización. No obstante habrá muchos comandos y sus opciones que, como podrán comprobar, son de uso tan frecuente que por repetición terminarán grabados en la memoria del lector.

El salvavidas se llama *man*, por manual. Como veremos posee la documentación de los comandos del sistema y también de una gran cantidad de funciones que forman parte de las *system calls*.

Para invocarlo es *man* y a continuación el nombre del comando. Recomendamos al lector principiante, en primer lugar tipear:

```
$ man man
```

Ahora sabemos como pedir ayuda. El *man* es una aplicación de uso muy frecuente. Lo recomendable es para cada comando que necesitemos utilizar listar su documentación en el *man* para ver las opciones disponibles y como aprovechar mejor sus posibilidades.

Comandos útiles

Como todo sistema operativo, Linux organiza el disco duro de nuestro computador en una estructura jerárquica de directorios (o carpetas), dentro de los cuales existen mas directorios, o archivos[1].

Cada usuario posee lo que Linux denomina un *Home Directory*. Aquí es donde aparecemos cada vez que efectuamos un login. Es nuestra plataforma de despegue por defecto. Generalmente la ruta está en /home/nombre_del_usuario. Este directorio se crea junto con el usuario en el sistema, ejemplo */home/juan, /home/luis*, */home/rober.*

Los usuarios, juan, luis, y rober cada vez que pasen el login ingresando su

1 Como veremos al abordar el tema File System, los archivos pertenecen a un grupo mas general de entidades lógicas denominadas nodos.

nombre (userID) y password, arrancarán en sus directorios "Home".

Como comprobar el directorio en el que nos encontramos?. Sencillo:

```
$ pwd
```

¿Que es $?. Se denomina *prompt* (índice), y solo es una indicación de que el shell está listo para recibir nuestros comandos. Puede modificarse su aspecto mediante procedimientos que no explicaremos aún, pero si el lector observa con detenimiento la figura 2, verá que en la ventana de fondo negro el prompt dice:

```
alejandro@notebook:~$
```

alejandro es mi *userID*, **notebook** es el nombre que le di a mi computador cuando instalé Linux, y **~** es una forma abreviada que nos proporciona Linux, para referenciar el *home directory* de nuestro usuario.

Recién logeados al sistema y sabiendo que *man* nos sacará de cualquier apuro, vamos a ver el contenido de nuestro *home directory*. Para listar el contenido de un directorio, el comando es *ls*.

Aparecerán entre otras entradas, una llamada *Desktop*. Corresponde a nuestro escritorio Gráfico y contiene los archivos que tenemos en el escritorio.

¿Como ingresamos?. Mediante el comando cd (**c**hange **d**irectory)[2]

```
alejandro@notebook:~$ cd Desktop/
alejandro@notebook:~/Desktop$
```

Observar el resultado en el *prompt*. Se va incluyendo la ruta de directorios para tener al usuario orientado acerca de donde se encuentra y evitar el uso del comando *pwd*.

Con *ls* podemos entonces inspeccionar los contenidos de cada directorio en el que nos situemos con *cd*. Sin embargo la salida del comando *ls* tipeado así solo, no es muy generosa.

Generalmente queremos conocer no solo que archivos hay en el directorio sino que además necesitamos conocer sus características. Y por lo general queremos ver todos los archivos, incluso los que empiezan con el carácter '.' (punto).

La forma mas general del comando *ls* que se utiliza en la práctica es *ls -las*. Tipeando *man ls* podrán enterarse del significado de las opciones 'l', 'a', y 's'.

Generalmente ordenaremos nuestro trabajo en directorios de modo que necesitamos poder crearlos. El comando mkdir, resuelve el problema:

2 El lector principiante habrá pensado "¡Igual que en Windows y DOS!". En realidad DOS y Windows tomaron este y varios otros comandos de shell de Unix, desarrollado en 1969, cuando aún no existían los microprocesadores, y que es la base de los sistemas operativos modernos ;-)

```
alejandro@notebook:~$ mkdir infoi
alejandro@notebook:~$
```

¿Nos equivocamos con el nombre?. No es problema. El comando *rm* (remove) puede borrar cualquier elemento del disco (a esta altura no debería ser necesario, pero cumplo en recordar que consultar *man* por cada comando es parte de la lectura de esta guía). Para los directorios, *rm* requiere de la opción -r (recursive) ya que borra el directorio y TODO su contenido.

```
alejandro@notebook:~$ rm infoi
rm: no se puede borrar «infoi»: Es un directorio
alejandro@notebook:~$ rm -r infoi
alejandro@notebook:~$
```

Ya está borrado. Observar que no pregunta si estamos seguros de querer borrarlo. Lo borra y ya.

```
alejandro@notebook:~$ mkdir info1
alejandro@notebook:~$
```

Ingresamos a nuestro directorio ***info1***, usando el comando *cd*. Luego examinamos su contenido, mediante *ls -las*.

El directorio está aún vacío ya que no hemos creado nada dentro de él. Sin embargo aparecen dos entradas, denominadas '.', y '..'[3].

La entrada '.' es una referencia al propio directorio. La prueba es que si tipeamos *cd.* no pasa nada, nos quedamos en el mismo directorio.

La entrada '..' es una referencia al directorio padre de ***info1***, es decir aquel en el que lo hemos creado. En este caso es nuestro *home directory*. Para comprobarlo:

```
alejandro@notebook:~/info1$ cd ..
alejandro@notebook:~$
```

Para operar con archivos los comandos cp (copiar) y mv (mover), nos permiten copiar un archivo de un lugar[4] a otro en el disco, o mover/renombrar un archivo.

Si no recordamos la ubicación de un archivo de trabajo como por ejemplo un programa fuente, el comando *find* nos puede resultar de utilidad:

```
alejandro@notebook:~$ find . -name Guia000-Linux.odt -print
./work/facu/Info I/Guias/Guia000-Linux.odt
```

3 ¿Otra vez pensó "¡Igual que Windows!"?. :-)

4 EN realidad el término correcto es ruta, o Path, pero aún o llegamos a este punto.

```
alejandro@notebook:~$
```

En general le indicamos como argumentos desde que punto del árbol de directorios queremos buscar (en nuestro caso '.' significa "desde aqui"), el nombre del archivo (-name .....), y que imprima en consola la salida (-print)

Este comando es el mas antiguo. Tarda un tiempo considerable debido a que busca en el disco en forma recursiva cada en cada directorio que dependa del punto de origen de la búsqueda.

En cambio locate, es un comando que usa bases de datos que mantiene el sistema operativo y es mas simple y rápido.

```
alejandro@notebook:~$ locate Guia000
/home/alejandro/work/facu/Info I/Guias/Guia000-Linux.odt
alejandro@notebook:~$
```

Observese que ni siquiera requiere el nombre completo del archivo, ni especificar nada mas que un patrón de búsqueda

Cuando se quiere saber en donde se ubica un archivo que contiene un binario ejecutable, existe un comando específico: *which*.

```
alejandro@notebook:~$ which ls
/bin/ls
alejandro@notebook:~$
```

Aunque en general los programas están en las siguientes rutas

/bin: Binarios, programas utilizados durante el booteo del sistema.

/usr/bin: Binarios de usuario, programas estándar disponibles para usuarios.

/usr/local/bin: Binarios locales, programas específicos de una instalación.

Como vemos el *shell* no tiene comandos internos (es decir incrustados en su propio código). Para los memoriosos, el *command.com* (*shell* del legendario MS-DOS), incluía un set de comandos internos que resolvían determinadas cuestiones. Por ejemplo, *dir* era el comando equivalente a ls en UNIX. A propósito... vaya a la consola y tipee *dir*. Verá que en Linux se han apiadado de los usuarios provenientes del mundo Windows/DOS.

Volviendo a las características del *shell*, en Unix las cosas se hicieron bien desde el inicio: el *shell* es un programa que interpreta ordenes, nada mas.... y nada menos. Y sus hijos, como Linux siguen la política inicial. Para mas pruebas tipee *which dir*.

Liste con *ls -las* ambos archivos binarios */bin/ls* y */bin/dir*, y compare sus tamaños.

Resumiendo algunas operaciones, vamos a traer a nuestro directorio info1, un

archivo bastante interesante.

Para ello usamos el comando *cp*

```
alejandro@notebook:~/info1$ cp /etc/passwd .
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 ene 24 12:24 .
4 drwx------ 92 alejandro alejandro 4096 ene 24 09:07 ..
4 -rw-r--r--  1 alejandro alejandro 1525 ene 24 12:24 passwd
alejandro@notebook:~/info1$
```

El archivo /etc/passwd contiene información de los diferentes usuarios que existen definidos en el sistema, sus UserID, y las contraseñas. ¿Como?¿las contraseñas? ¿y la integridad y la seguridad de los sistemas Unix?. Calma. Las contraseñas están cifradas. En este archivo hay una x o un * en ese campo. Además las contraseñas se encriptan mediante un algoritmo que tiene la particularidad de generar una clave única en el sentido directo, pero en el inverso (es decir tratando de descifrar la clave) arroja mas de un resultado, y todos diferentes del original. De modo que al logearnos, cuando introducimos la clave, la única forma de validarla es encriptarla y comparar los valores encriptados buscando que sean iguales. Una vez encriptada nuestra password, nadie la podrá descifrar.

¿Como cambio mi contraseña?. Con el comando *passwd*. Primero pide la contraseña actual, y luego la nueva en dos oportunidades para asegurarnos que la tipeamos correctamente.

En ambos casos (es decir, tanto para el archivo */etc/passwd*, como para el comando *passwd*) se puede consultar *man*, para mas información. Si el lector prestó atención al comando *man man*, utilizado como primer ejemplo, habrá visto que la información en *man* está dividida en secciones cada una identificable con un número. Esto es porque muchas veces los comandos del *shell* llevan el mismo nombre de su archivo de configuración, cuya documentación es de interés, o de una *system call*. La sección 1 corresponde a los comandos de shell y la 5 a los archivos de configuración. Queda para el lector probar el efecto de los siguientes comandos:*man passwd*, *man 5 passwd*, y *man -a passwd*. Cualquier duda ejecutar nuevamente *man man* y esta vez leerlo mas detenidamente ;-).

Aclarado este punto accesorio, sigamos en las bases. Para mostrar el contenido de un archivo en la consola, el comando es *cat*. Si el archivo es binario, la salida es un jeroglífico, y si es texto, podemos leerlo.

Cuando el contenido del archivo es muy extenso tal vez convenga utilizar el

comando paginador *more*[5].

En ambos casos se tipea el comando seguido del nombre del archivo. Cualquier duda... *man*.

UNIX posee desde sus primeros tiempos una colección de comandos denominados filtros, debido a que pueden detectar patrones en el contenido de archivos o de flujos de datos en general. El mas simple es *grep*. El siguiente comando presenta solo las líneas del archivo passwd copiado en nuestro directorio info1, que contienen la palabra root.

```
alejandro@notebook:~/info1$ grep root passwd
root:x:0:0:root:/root:/bin/bash
alejandro@notebook:~/info1$
```

El comando *wc* seguido del nombre de un archivo presenta en la línea siguiente de la consola la cantidad de líneas, palabras y caracteres (en ese orden) que contiene un archivo. Es sumamente útil para analizar archivos de texto. Para archivos binarios es un tanto incierto el resultado. Continuando con nuestra copia en info1 del archivo passwd,

```
alejandro@notebook:~/info1$ wc passwd
  33   49 1525 passwd
alejandro@notebook:~/info1$
```

Se interpreta que el archivo ~/info1/passwd tiene 33 líneas, 49 palabras, y 1525 caracteres (o bytes). Los resultados presentados pueden obtenerse parcialmente con las opciones adecuadas. ¿Cuales son esas opciones?... *man wc*. Repetir el comando con -l, -w, y -c.

File System

Generalidades

En este punto es necesario hacer una pausa en la descripción de comandos básicos y pasar a entender un concepto central.

Un *file system* es un conjunto de políticas definidas para la organización de la información en archivos dentro de un medio de almacenamiento, de modo de permitir definir su ubicación dentro de ese medio y posibilitar el acceso a los datos contenidos por éstos de manera simple y transparente a los detalles del hardware.

El *file system manager* es una pieza de software de un sistema operativo que implementa las políticas definidas para el *file system*.

5 A esta altura del presente trabajo y luego de las reiteradas explicaciones históricas de UNIX, el autor tiene la legítima ilusión de que el lector ya ha dejado de exclamar para sí "¡Como en Windows!" ;-)

Los *file system* se definen siempre sobre los dispositivos de almacenamiento masivo de información, como lo son los discos duros, los CD's, DVD's, por ejemplo.

Los discos están compuestos de sectores cuyo tamaño en bytes es un número potencia de dos (típicamente 512, 1024, 2048, o 4096).

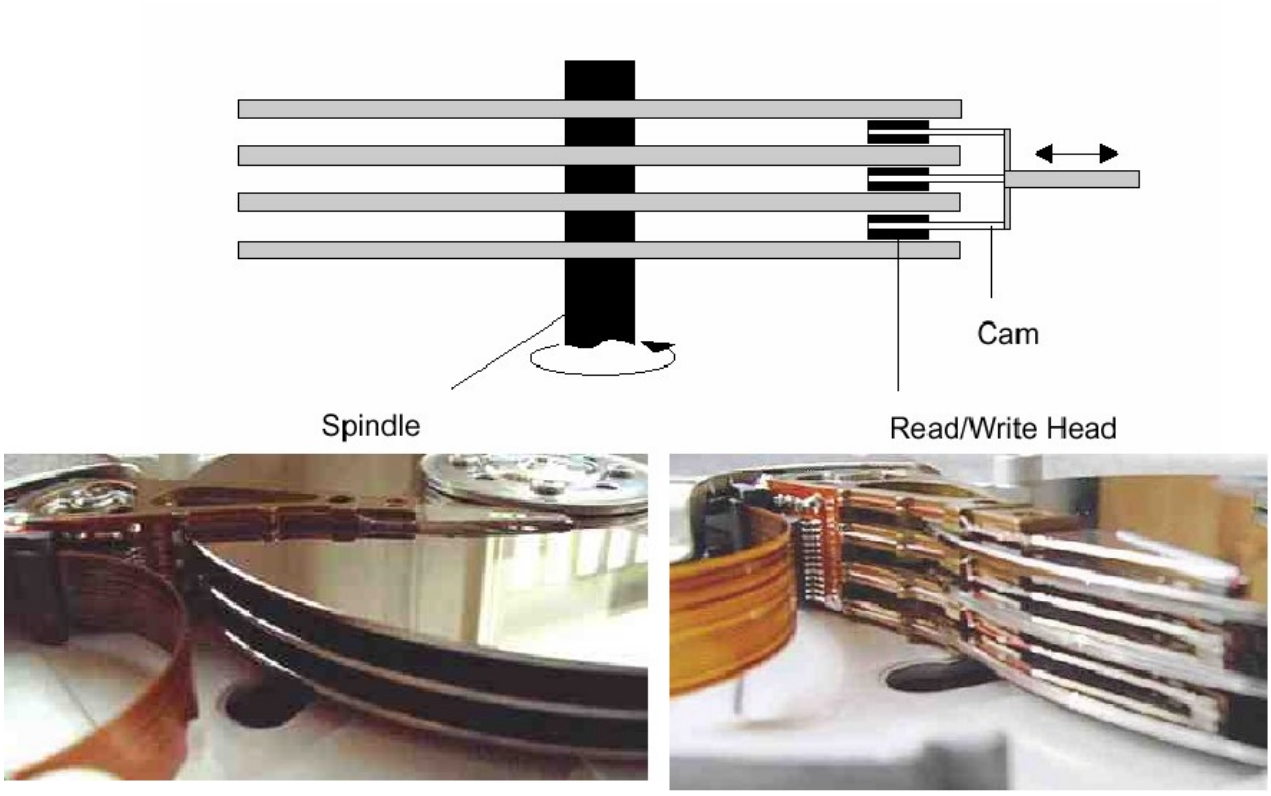

Fig3. Vista de un disco rígido

En la Figura 3 podemos ver su geometría. Es un armado de platos generalmente de aluminio, vidrio u otro material rígido, al que se le impregna una capa de material magnético en el cual se graba en forma permanente la información mediante traducción de '1's y '0's a cambios de orientación magnética (Norte-Sur). Cada plato posee material magnético en ambas caras, y por lo tanto se requiere un lector magnético de cada lado. El conjunto de pares lectores para cada plato se fijan a un único eje y avanzan en conjunto.

Al tener un mecanismo de acceso mecánico, el tiempo que se demora en obtenerse la información es mayor, comparado con el de un chip de memoria, ya que éste último está íntegramente construido de transistores MOS FET, cuyo tiempo de acceso es muchísimo menor.

El armado de lectores avanza de forma transversal a los discos (desde el norde exterior al borde interior o viceversa)

Cada vez que el armado de cabezas se posiciona en el lugar correcto en el que está la información, la cabeza correspondiente se aproxima hacia la superficie del disco para leer la información magnética y convertirla en pulsos eléctricos

que representen finalmente la secuencia de '1's y '0's que compone la información almacenada en código binario.

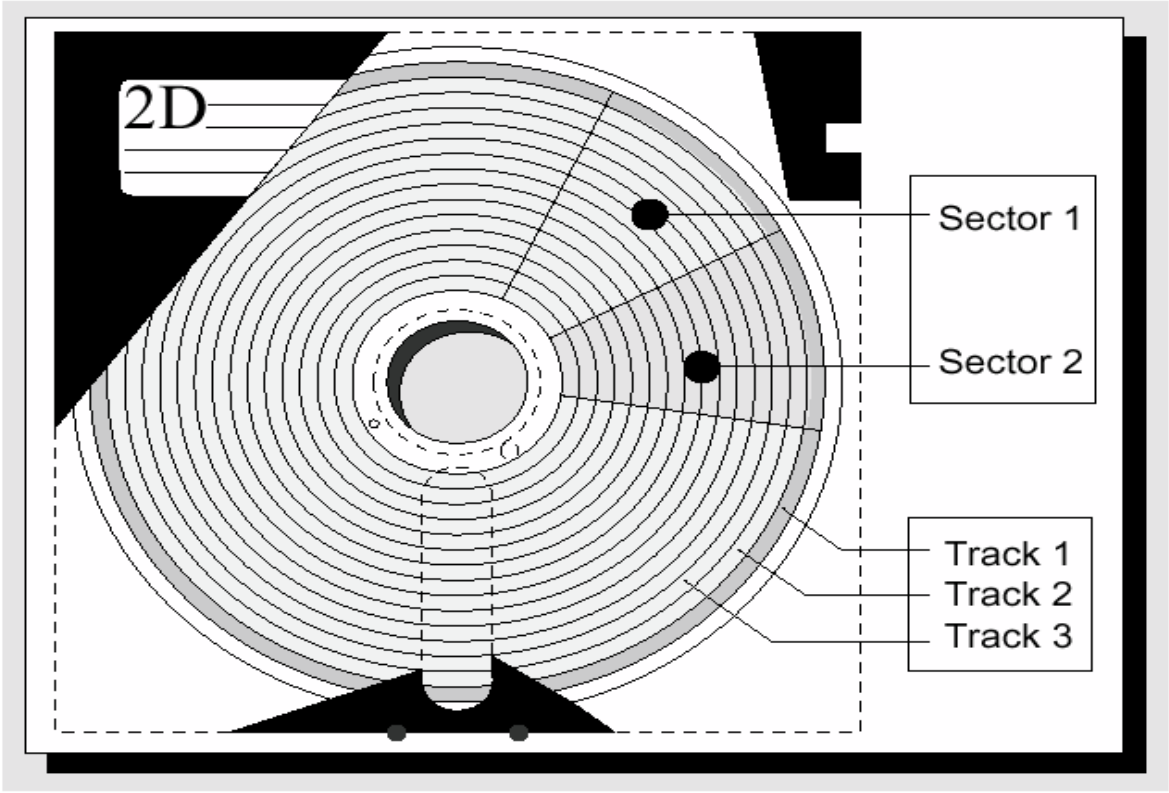

Fig. 4. Organización de un disco

La Figura 4, muestra como está organizada la información dentro de cada cara de cada disco.

Cada Cara se divide en coronas circulares concéntricas, conocidas como pistas (del inglés 'track).

A su vez el disco se divide radialmente de tal manera que cada pista queda compuesta de tramos denominados sectores.

Por lo tanto para que el hardware controlador de este complejo dispositivo nos pueda dar la información que necesitamos, requiere que se le especifiquen las coordenadas de la información en términos de la geometría del disco, esto es:

cabeza: es decir el dispositivo lector que debe activarse para leer la información magnética, o dicho de otro modo, debemos indicarle en que cara de que plato queremos leer.

Pista: una vez determinada la coordenada anterior el controlador debe saber dentro de esa cara de ese plato, en que pista está la información. Esto hace que pueda determinar hasta donde desplazar el armado de cabezas lectoras.

Sector: Una vez situados en la cara del plato que contiene la información, y dentro de esta cara la pista, se debe definir cual de los *n* sectores en que se

divide la pista es el que hay que transferir desde los circuitos de control del disco a la memoria del sistema.

Entonces, si nos quedamos con este nivel de manejo, el acceso a disco es un asunto de enorme complejidad. ¿Alguien podría averiguar fácilmente las coordenadas cabeza-pista-sector de cada archivo que tiene guarado en su disco rígido?. Sería una locura.

Por tal motivo se organiza la información en forma de archivos. Por ejemplo Guia000-Linux.odt, es el archivo en el que me encuentro escribiendo esta guía. No conozco, (i ni me hace falta !) la ubicación en el disco, en términos de las coordenadas físicas (así se lama en la jerga a cabeza-pista-sector).

Alguien nos tiene que ayudar. El File System Manager.

El *file system manager* es el responsable de organizar esos sectores en archivos, directorios, enlaces, etc, y mantener actualizada la base de datos de los sectores que corresponden a cada archivo, directorio, enlace, etc. De este modo provee una interfaz de mas alto nivel, y por ende mas fácil de manejar.

Por lo tanto cuando formateamos un disco, no estamos haciendo otra cosa que implementarle esta organización, dejando todo el espacio de almacenamiento vacío y listo para ser ocupado de acuerdo a las reglas establecidas para el *file system* seleccionado.

Su flexibilidad es tal que el medio puede ser accedido a través de una red de datos (*network file system*), o se construye en forma dinámica directamente en la memoria RAM del computador en el momento en que arranca el Sistema Operativo y nunca reside en el disco, (por eso se debe construir cada vez que se arranca el Sistema Operativo), en cuyo caso se trata de un *file system* virtual (conocidos en la jerga como *virtual file systems*).

Un *file system* diseñado para hostearse en un disco, puede también hostearse en una memoria Flash. Aunque por las características físicas de estos dispositivos no es óptimo su funcionamiento, ya que la memoria para ser modificada debe antes ser borrada, y no tiene un acceso secuencial como los discos, se suelen usar de todos modos los mismos *file systems* que en los discos para una mejor homogeneidad de gestión por parte del Sistema Operativo. No obstante existen *file system* específicos para memorias Flash.

Cada elemento de un *file system*, (típicamente cada archivo, aunque como veremos el archivo si bien es el elemento mas común de un *file system* no es necesariamente el único elemento posible), recibe un nombre que lo identificará en el lenguaje familiar para los seres humanos. En general los nombres se conforman con caracteres alfanuméricos, y especiales y de puntuación, aunque cada sistema operativo utiliza determinados caracteres ya sea para propósitos especiales como en casos en los que el nombre de un archivo es argumento de un comando de *shell* por lo cual en cada sistema operativo existe un grupo de caracteres especiales que no pueden utilizarse en los nombres de los archivos.

Además del nombre, un *file system* almacena para cada elemento un conjunto de información adicional, que se denomina *Metadata*. Por un lado resulta útil almacenar el tamaño de un elemento del *file system*, en especial si este elemento es un archivo. Para tal fin se puede guardar la cuenta exacta de bytes que componen el mismo o la cantidad de bloques de disco que ocupa (observar que no tienen necesariamente que coincidir ambos valores). Por otra parte es útil almacenar los *timestamps*[6] de creación, y última modificación de cada archivo. Otro dato de utilidad es el tipo de elemento, es decir, si se trata de un archivo, o de un device, o de un directorio, o de un enlace, etc. También se incluye en la *Metadata* información referente a que usuario es dueño del archivo, a que grupos de usuarios es accesible y para que propósitos (solo lectira, lectura/escritura, ejecución, etc). En términos generales estos son los atributos mas comunes en la *Metadata*, sin embargo cada *file system* puede poseer atributos particulares, como por ejemplo, el checksum del contenido del elemento como un control de integridad del mismo.

File Systems y el Sistema Operativo.

Por lo general cada Sistema Operativo posee uno o mas *file systems* propios. Algunos sistemas operativos (como Windows) solo trabajan con sus propios file systems, y de haber en el disco duro del computador particiones con otros file systems, simplemente las ignoran como i no estuviesen. Otros en cambio como Linux leen prácticamente de cualquier formato de *file system* conocido, y ampliamente difundido.

Windows trabaja con los file system FAT y NTFS. FAT es el heredado de DOS, y tiene tres implementaciones FAT12, FAT 16 y FAT 32.

FAT 12, es para disquettes, y es un standard a tal punto que los disquettes bajo Linux utilizan por defecto FAT12.

FAT 16 es una especie en extinción. Diseñado para dar soporte en DOS a discos duros cuando estos aparecieron, hoy ha perdido vigencia ya que soporta discos de 512 Mbytes máximo de tamaño.

FAT 32 fue la respuesta de emergencia a esta limitación de FAT 16, pero como toda evolución de una mala idea está predestinada a seguir los pasos de sus ancestros. Su principal limitación es no soportar archivos de mas de 4Gbytes de tamaño. Una imagen de DVD tiene típicamente entre 4,3 y 4,5 Gbtes.

Hoy se utiliza NTFS en cualquier instalación de Windows ya que fue un *file system* pensado sin restricciones de compatibilidad, y es sumamente sólido y confiable, y no posee las restricciones mencionadas en los otros sistemas.

Linux tiene la familia ext*, que lleva hasta ahora tres versiones: ext2, ext3, y ext4. Esta guía está siendo redactada en un equipo con ext3 y funciona con suma robustez y confiabilidad.

6 Un timetamp es una secuencia de caracteres que conforman la fecha y hora de ocurrencia de un evento.

Además este equipo posee una partición con Windows XP, formateada en NTFS, y es perfectamente visible desde Linux. Sino, observe en la figura el directorio, del navegador konqueror y vea si no le resulta familiar.

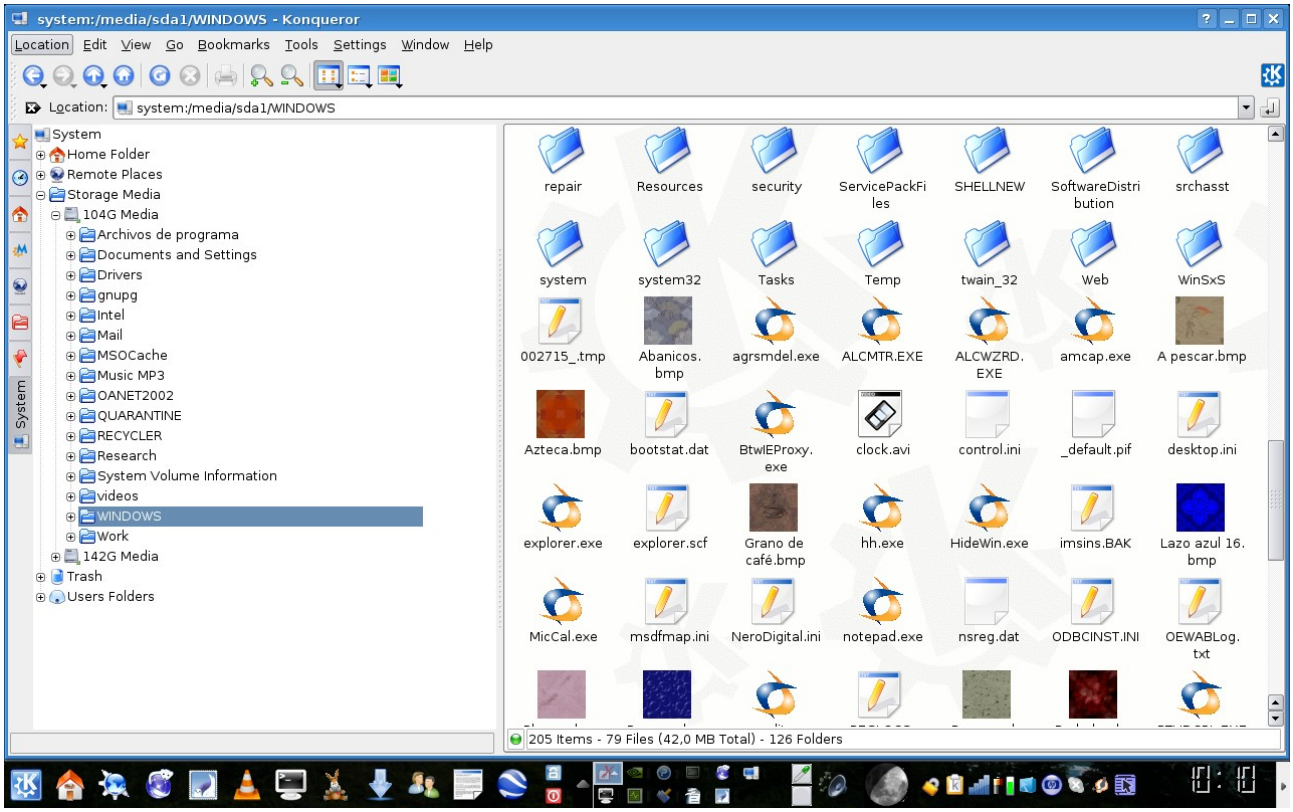

Fig. 5. Vista del directorio C.\Windows de una partición NTFS desde Linux.

Las Mac corren por lo general el Sistema Operativo MacOS, y su file system nativo es HFS plus.

También en los mundos UNIX de diversos fabricantes de hardware se pueden encontrar diversos tipos de *file systems*.

Visualización de los datos del File System en Linux

A la luz de estas definiciones acerca de *file systems*, M*etadata*, etc, examinemos el contenido de algunos directorios de interés.

```
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 ene 24 12:24 .
4 drwx------ 92 alejandro alejandro 4096 ene 24 09:07 ..
4 -rw-r--r--  1 alejandro alejandro 1525 ene 24 12:24 passwd
alejandro@notebook:~/info1$
```

Ya nos referimos oportunamente al significado de las entradas '.' y '..'. Sin embargo al igual que la entrada passwd poseen una cantidad de información

que ahora podemos analizar.

En principio, las entradas '.' y '..' aparecen gracias a que hemos incluido la opción *-a* (all) en el comando *ls*, que indica que se deben listar todas las entradas del directorio, incluidas estas dos. De otro modo no son presentadas. Para comprobarlo, queda a cargo del lector ejecutar en el mismo directorio en cuestión el comando *ls -ls*. Es decir omitiendo la opción -a y dejando solamente las opciones *-l* (long), y *-s (size)*.

Para cada entrada podemos ver de izquierda a deracha:

- El tamaño en bloques de 1024 bytes del elemento. Este campo es el único debido a la opción -s, el resto corresponde a la opción -l.
- Tipo de elemento del file system. Los archivos son el tipo mas común, de modo que no llevan ningún carácter en particular. Por ello este campo para el archivo passwd tiene el carácter '-'. Si observamos las otras dos entradas vemos que este campo tiene el carácter 'd', por Directory. Hay otros tipos de elementos que veremos en ejemplos posteriores.
- Los permisos del elemento. En este caso los permisos se componen de tres campos. Siempre de izquierda a derecha: permisos para el usuario dueño del archivo, permisos para el grupo al que pertenece el usuario dueño del archivo, y permisos para el resto de los usuarios. Como puede verse cada campo tiene tres caracteres, los cuales siempre yendo de izquierda a derecha son: 'r' por Read para denotar si el archivo puede ser leído, 'w' por Write para permitir su modificación, y 'x' por eXecutable. Cuando un permiso no está disponible ya sea para el usuario, para el grupo de este usuario, o para otros, el lugar correspondiente lleva el carácter '-'. Como puede verse el usuario dueño, los usuarios miembros del grupo del usuario dueño y los otros usuarios (el resto) pueden tener diferentes permisos de acceso al archivo. Solo el usuario dueño puede modificar los permisos de sus archivos. En todos los Sistemas Operativos que se precien de tales, existe un superusuario que es quien tiene el control de todo el sistema, y por lo tanto accede a todos los recursos, procesos, y archivos. En Linux se llama root. Este usuario tiene respecto de todos los archivos las mismas atribuciones que los usuarios dueños respectivos.
- El número de enlaces (*linksmas adelante explicaremos el concepto de enlace y sus comandos asociados.*
- El *UserID* del propietario, y el del grupo al que pertenece (*GroupID*).
- El tamaño (en bytes),
- El *timestamp* correspondiente a la última modificación del elemento,
- El nombre del elemento.

Como se puede observar presenta buena parte de la Metadata imprescindible para tener a mano de cada entrada, y su nombre.

En el caso de Linux, el nombre es solo eso: no importa si hay un punto que lo separa en dos partes. La extensión que usan Windows y DOS, en Linux no existe. La asociación que hace el sistema de archivos típicos, como por ejemplo un archivo fuente de lenguaje C, es parseando el nombre desde derecha a izquierda, hasta encontrar un punto ('.').

Veamos el siguiente listado:

```
alejandro@notebook:~/info1$ ls -ls /dev/pts
total 0
0 crw------- 1 alejandro tty 136,  0 feb  5 15:34 0
0 crw------- 1 alejandro tty 136,  1 feb  5 06:56 1
0 crw------- 1 alejandro tty 136, 10 feb  5 06:56 10
0 crw------- 1 alejandro tty 136, 11 feb  5 06:56 11
0 crw------- 1 alejandro tty 136, 12 feb  5 06:56 12
0 crw--w---- 1 alejandro tty 136, 13 feb  5 06:56 13
0 crw--w---- 1 alejandro tty 136, 14 feb  5 06:56 14
0 crw------- 1 alejandro tty 136, 15 feb  5 06:57 15
0 crw--w---- 1 alejandro tty 136, 16 feb  5 06:57 16
0 crw--w---- 1 root      tty 136, 17 feb  5 06:58 17
0 crw------- 1 alejandro tty 136,  2 feb  5 06:56 2
0 crw------- 1 alejandro tty 136,  3 feb  5 06:56 3
0 crw------- 1 alejandro tty 136,  4 feb  5 06:56 4
0 crw------- 1 alejandro tty 136,  5 feb  5 06:56 5
0 crw------- 1 alejandro tty 136,  6 feb  5 06:56 6
0 crw------- 1 alejandro tty 136,  7 feb  5 06:56 7
0 crw------- 1 alejandro tty 136,  8 feb  5 06:56 8
0 crw------- 1 alejandro tty 136,  9 feb  5 06:56 9
alejandro@notebook:~/info1$
```

Las entradas del listadas, corresponden a las pseudo-terminales del sistema. Si observamos el Tipo de archivo ha cambiado respecto del listado anterior. La 'c' con que arranca cada línea denota que se trata de un dispositivo de Caracteres[7].

Por otra parte en los campos en los que antes aparecía el tamaño del archivo ahora aparecen dos números separados por una ','. Estos dos campos se conocen como Número Mayor y Número Menor (de izquierda a derecha

7 Los dispositivos de carácter son aquellos que intercambias byte a byte con el sistema de E/S.

respectivamente), y sirven para el manejo por parte del Sistema Operativo de estos dispositivos con el mismo grupo de funciones que se utilizan para tratar archivos. Gracias a esto el paradigma "everything is a file" permite en Linux el manejo de una serie mas que interesante de dispositivos con las mismas *system calls* que utilizan los archivos comunes y corrientes.

Veamos este otro listado:

```
alejandro@notebook:~/info1$ ls -ls /dev/sda*
0 brw-rw---- 1 root disk 8, 0 feb  5 04:54 /dev/sda
0 brw-rw---- 1 root disk 8, 1 feb  5 10:09 /dev/sda1
0 brw-rw---- 1 root disk 8, 2 feb  5 04:54 /dev/sda2
0 brw-rw---- 1 root disk 8, 3 feb  5 04:54 /dev/sda3
0 brw-rw---- 1 root disk 8, 5 feb  5 04:54 /dev/sda5
```

Hemos presentado un conjunto correspondiente a las particiones del disco duro que tiene para manejar el sistema. Dentro de la enorme cantidad de dispositivos que contiene el directorio, hemos logrado mediante el carácter '*' que significa en términos de un nombre de archivo algo así como "lo que sea", presentar solo éstos que son de nuestro interés. En nuestro caso el comando se entiende como listar todos los nombres que comiencen con *sda* y que sigan con "lo que sea".

El caracter '*' es entonces uno de los que mencionamos al referirnos a la composición del nombre de un elemento del *file system*, como caracteres que no pueden usarse dentro del nombre de un archivo u otro elemento. Las razones son bastante obvias a partir de este ejemplo.

Obsérvese que el primer carácter a la izquierda de cada fila del listado (correspondiente al campo "tipo de elemento") es ahora una 'b'. Esto es debido a que los discos se definen como dispositivos de Bloque[8].

A continuación, algunos otros ejemplos interesantes para incorporar mas conceptos.

```
alejandro@notebook:~/info1$ ls -ls /dev/scd*
0 brw-rw---- 1 root cdrom 11, 0 feb  5 04:54 /dev/scd0
alejandro@notebook:~/info1$ ls -ls /dev/cd*
0 lrwxrwxrwx 1 root root 4 feb  5 04:54 /dev/cdrom -> scd0
0 lrwxrwxrwx 1 root root 4 feb  5 04:54 /dev/cdrw -> scd0
alejandro@notebook:~/info1$ ls -ls /dev/sdc*
```

8 Los dispositivos de bloque intercambian información con el sistema de E/S de a bloques de un número de bytes potencia de 2. Típicamente se utilizan 512 bytes 1024, 2048, o 4096. En el caso de los discos duros y floppys, es 512 ya que coincide con el tamaño de los sectores en los que se dividen estos discos internamente. El resto de los valores los emplean los discos ópticos.

En el segundo comando vemos que las entradas tienen una 'l' en el primer carácter de la entrada. Así se denotan los links, o enlaces.

Que son los *links*?. En Unix existen dos conceptos de *link* o 'enlace', llamados usualmente enlace duro o físico (*hard link*) y enlace blando o simbólico (*soft link*). Un *hard link* es simplemente un nombre para un archivo, o dispositivo, o genéricamente un elemento del *file system.* Unelemento del *file system* puede tener varios nombres. Se borra del disco solamente cuando se elimine el último nombre. El número de nombres lo muestra el comando ls con la opción -l, a la derecha de los permisos. No existe en UNIX el concepto de nombre 'original', sino que todos tienen la misma categoría. Usualmente, pero no necesariamente, todos los nombres de un archivo se encuentran dentro del *file system* que también contiene sus datos.

Un *soft link* (o enlace simbólico, o acceso directo) es un pequeño archivo especial que contiene la ruta (*path*) al archivo. Por lo tanto, los *soft links* pueden apuntar a archivos en diferentes *file systems* (incluso montados vía NFS desde computadores diferentes), y no tienen por qué apuntar a ficheros que existan realmente. Cuando se accede a ellos con las *system calls open ()* o *stat ()*, el kernel reemplaza una referencia a un *soft link* con una referencia al archivo nombrado en el nombre del *path* contenido por el *soft link*. Sin embargo, con las *system call rm ()* y *unlink ()* se borra el mismo enlace, no el archivo al cual apunte.

Para crear un link el comando es *ln*. La opción *-s* especifica que se trata de un *soft link*. Por default (sin opciones es un *hard link*). El siguiente comando crea un *hard link* a nuestro archivo copia del *passwd*, dentro del directorio info1.

```
alejandro@notebook:~/info1$ ln passwd password
alejandro@notebook:~/info1$ ls -las
total 16
4 drwxr-xr-x  2 alejandro alejandro 4096 feb  7 05:25 .
4 drwx------ 92 alejandro alejandro 4096 feb  6 20:35 ..
4 -rw-r--r--  2 alejandro alejandro 1525 ene 24 12:24 passwd
4 -rw-r--r--  2 alejandro alejandro 1525 ene 24 12:24 password
alejandro@notebook:~/info1$
```

Observar la salida del comando *ls -las* posterior: ambos elementos figuran con 2 en el campo cantidad de enlaces. Esto significa que los archivos *passwd*, y *password* son dos entradas diferentes en el *file system* pero con un punto importante en común: apuntan a la misma área del disco. Prueba de ello es que ambos tiene la misma Metadata En especial llama la atención el *timestamp*.

Vamos a borrar el archivo "original", *passwd*.

```
alejandro@notebook:~/info1$ rm passwd
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 feb  7 05:27 .
4 drwx------ 92 alejandro alejandro 4096 feb  6 20:35 ..
4 -rw-r--r--  1 alejandro alejandro 1525 ene 24 12:24 password
alejandro@notebook:~/info1$
```

Al borrar el archivo *passwd*, nos queda el *password* con 1 en el campo cantidad de enlaces, y se mantiene el contenido.

Esto permite tener múltiples referencias a la misma zona de disco sin replicar la información, es decir manteniendo una sola copia de la misma.

Para crear un *soft link*, hacemos lo siguiente:

```
alejandro@notebook:~/info1$ ln -s password passwd
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 feb  7 05:35 .
4 drwx------ 92 alejandro alejandro 4096 feb  6 20:35 ..
0 lrwxrwxrwx  1 alejandro alejandro    8 feb  7 05:35 passwd -> password
4 -rw-r--r--  1 alejandro alejandro 1525 ene 24 12:24 password
```

Aquí observamos algunas diferencias con el caso anterior. En primer lugar, el tipo de entrada de *passwd* se alteró, apareciendo el carácter *'l'* para denotar que se trata de un *soft link*. Y la cantidad de enlaces de ambas entradas permanece en 1 ya que de hecho son estructuras diferentes: *passwd* es un archivito que apunta a password, es decir que contiene su ruta absoluta: */home/alejandro/info1/password*. Prestar también atención al cambio del timestamp.

Veamos a continuación los comandos *chmod*, *chown*, *chgrp*.

Con *chmod* podemos cambiar los permisos de un archivo para el que tengamos permiso de acceso. Cada usuario puede cambiar permisos de aquellos archivos de los que es dueño (owner).

```
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 feb  7 05:35 .
4 drwx------ 92 alejandro alejandro 4096 feb  7 09:50 ..
0 lrwxrwxrwx  1 alejandro alejandro    8 feb  7 05:35 passwd -> password
4 -rw-r--r--  1 alejandro alejandro 1525 ene 24 12:24 password
```

```
alejandro@notebook:~/info1$ chmod g+w password
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 feb  7 05:35 .
4 drwx------ 92 alejandro alejandro 4096 feb  7 09:50 ..
0 lrwxrwxrwx  1 alejandro alejandro    8 feb  7 05:35 passwd -> password
4 -rw-rw-r--  1 alejandro alejandro 1525 ene 24 12:24 password
```

La forma de uso es "ugoa" (User, Group, Others, All), seguido de '+' o '-' según se quiera agregar o quitar ese permiso respectivamente. En nuestro caso como hemos visto, solo hemos agregado permiso de escritura para los usuarios del mismo grupo del owner.

No opera sobre los *soft links* ya que al ser un mero puntero a un archivo (los *shortcuts* de Windows tomaron este elemento como idea central), los permisos son los del archivo al que apunta.

Otra forma de trabajar con *chmod* es asumir a los permisos como tres números octales consecutivos en donde cada bit octal se refiere al bit de permisos correspondiente. Si el permiso está habilitado el bit vale 1 y si no lo está el bit vale 0. Por ejemplo en el listado último, el archivo *password* quedó con permisos 664. De modo que para restablecer los permisos al estado previo otra forma alternativa de *chmod* es:

```
alejandro@notebook:~/info1$ chmod 644 password
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 feb  7 05:35 .
4 drwx------ 92 alejandro alejandro 4096 feb  7 09:50 ..
0 lrwxrwxrwx  1 alejandro alejandro    8 feb  7 05:35 passwd -> password
4 -rw-r--r--  1 alejandro alejandro 1525 ene 24 12:24 password
alejandro@notebook:~/info1$
```

Para cambiar el grupo que puede acceder a un archivo el comando es chgrp

```
alejandro@notebook:~/info1$ chgrp root password
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 feb  7 05:35 .
4 drwx------ 92 alejandro alejandro 4096 feb  7 09:50 ..
0 lrwxrwxrwx  1 alejandro alejandro    8 feb  7 05:35 passwd -> password
4 -rw-r--r--  1 alejandro root      1525 ene 24 12:24 password
```

Ahora los usuarios del grupo *alejandro* ya no tendrán acceso a *password*. En su lugar los usuarios del grupo *root* sí accederán.

Por su parte el comando *chown* permite cambiar el usuario dueño y el grupo a la vez. Siempre que se escriba *userID:groupID*. Si se escribe solo un nombre sin el separador ':', el comando asume que el argumento es el *userID*.

```
alejandro@notebook:~/info1$ chown alejandro:alejandro password
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 feb  7 05:35 .
4 drwx------ 92 alejandro alejandro 4096 feb  7 09:50 ..
0 lrwxrwxrwx  1 alejandro alejandro    8 feb  7 05:35 passwd -> password
4 -rw-r--r--  1 alejandro alejandro 1525 ene 24 12:24 password
```

Veamos lo que ocurre cuando se intenta cambiar el UserID del archivo.

```
alejandro@notebook:~/info1$ chown usuario password
chown: cambiando el propietario de «password»: Operación no permitida
alejandro@notebook:~/info1$ sudo chown usuario password
[sudo] password for alejandro:
alejandro@notebook:~/info1$ ls -las
total 12
4 drwxr-xr-x  2 alejandro alejandro 4096 feb  7 05:35 .
4 drwx------ 92 alejandro alejandro 4096 feb  8 07:33 ..
0 lrwxrwxrwx  1 alejandro alejandro    8 feb  7 05:35 passwd -> password
4 -rw-r--r--  1 usuario   alejandro 1525 ene 24 12:24 password
```

Como vemos el cambio del owner de un archivo no está permitido para usuarios comunes. Solo el superusuario (*root*) puede hacerlo.

Esto lo vemos claramente en el comando *sudo* (SUperuser DO). Este comando permite ejecutar con los privilegios de *root* el comando que continúa en la misma línea. Para que un usuario pueda ejecutar sudo, debe estar definido en el archivo */etc/sudoers* que solo puede administrar el usuario *root*. Queda para el lector recorrer *man* sudo y *man sudoers* para ampliar.

Respecto del *timestamp* el comando que permite cambiar las fechas y horas de creación y modificación de los archivos es *touch*. Queda para el lector su lectura (*man touch*), ya que no es muy habitual su uso, salvo si por accidente se modificara el reloj del sistema y la fecha y hora del archivo es vital para alguna aplicación.

Para revisar el porcentaje de uso de los *file sistems*, el comando df es el adecuado.

```
alejandro@notebook:~/info1$ df
S.ficheros          Bloques de 1K   Usado    Dispon Uso% Montado en
/dev/sda2             136954892 127156960   2840984  98% /
tmpfs                   1035800         8   1035792   1% /lib/init/rw
udev                      10240       104     10136   2% /dev
tmpfs                   1035800         0   1035800   0% /dev/shm
/dev/sda1             102398276  97486844   4911432  96% /mnt/windows
```

Si tuviese inconvenientes por utilizar algún *file system* de las versiones System V de UNIX, la opción *-v*, salva las compatibilidades y provee esta misma salida del comando anterior.

Estructura de directorios en Linux

Cuando observamos el *file system* de un sistema UNIX y lo recorremos, si previamente estamos habituados al universo Windows, percibimos ciertas diferencias.

Por eso es necesaria alguna mínima explicación de como está conformado un *file system* estándar antes de seguir adelante.

The Directory Structure in Unix & Linux are a unified Directory Structure where in all the directories are unified under the "/" Root file system. Irrespective of where the File System is physically mounted all the directories are arranged hierarchically under the Root file system.

La estructura de Directorios en Linux sigue los lineamientos de "Filesystem Hierarchy Structure (FHS)" establecidos por el Free Standards Group. No obstante es oportuno aclarar que de acuerdo a la distribución que encontremos, es posible notar alguna que otra desviación de este estándar.

Veremos a continuación los principales directorios que existen y cual es su propósito o contenido, bajo el *File System* Jerárquico de Linux. La descripción aplica a las distribuciones Debian y Ubuntu. Si el lector lo compara con otras distribuciones puede encontrar eventualmente alguna pequeña diferencia con la estructura siguiente.

Como primer cuestión se observa que el carácter '/' es el conector entre directorios. Esta convención fue introducida por UNIX en 1969. Cuando aparece el DOS en 1980, copia el concepto jerárquico y muchísimos otros conceptos y elementos para manejar el file system FAT 12 con el que se inició. Sin embargo adoptó como conector la otra barra '\'. Obviamente Windows continuaría con esta notación, con lo cual ahora el original parece estar al revés del universo. Paradojas de un éxito comercial.

"/" Root

La estructura de Directorios comienza con el Root file system "/", y se reconoce

como la raíz de toda la estructura de directorios.

/boot

Contiene los archivos del Boot loader incluyendo Grub o Lilo, el Kernel, el script *initrd* que es el inicializador general del sistema, y los archivos de configuración system.map.

/sys

Contiene archivos relacionados con el Kernel, el Firmware, y el sistema en general.

/sbin

Contiene los binarios esenciales para la Administración del Sistema (sbin deriva de System BIN), que en general son herramientas para operación y performance del Sistema Operativo.

/bin

Contiene los binarios esenciales para uso del sistema por arte de los usuarios: cat, ls, cp etc.

/lib

Contiene los archivos de librerías para todos los binarios contenidos en los directorios /sbin y /bin.

/dev

Contiene archivos esenciales para el sistema que permiten acceder a los Device Drivers.

/etc

Contiene archivos de configuración fundamentales del Sistema Operativo, como por ejemplo la configuración del sistema de red, y también de aplicaciones importantes como el servidor X para implementación de las interfaces gráficas de usuario por ejemplo.

/home

Todos los directorios home de los usuarios "cuelgan" de este directorio, a excepción del usuario *root* cuyo directorio home está en /root. El home de cada usuario contiene archivos de configuración personalizados para ese usuario, y settings como por ejemplo el archivo .profile.

/media

Es un directorio en el que se montan dispositivos removibles como Flash memories, pen drives, cámaras fotográficas, teléfonos celulares 2G y 3G, y además algunos dispositivos internos del computador tales como discos flexibles (floppy disk), CD's, y DVD's.

/mnt

Es un punto de montaje[9] genéricos para file systems temporarios.

/opt

En Linux se reserva para instalar Paquetes de Software opcionales (opt proviene de OPTional).

/usr

User data directory. Contiene aplicaciones y utilidades específicas para los usuarios. Existe una muy importante cantidad de fle systems que pueden montarse en Linux'. En ellos nuevamente podemos encontrar directorios bin, sbin, y lib. Se montan a partir de aquí en donde además pueden tener un directorio include para disponer allí de todos los archivos include que necesitan.

/usr/sbin

Contiene binarios del sistema y utilidades de red, que no son ni críticos ni esenciales.

/usr/bin

Contiene binarios para comandos de usuarios, que no son ni críticos ni esenciales.

/usr/lib

Aquí están los archivos que contienen las librerías de código que necesitan los archivos binarios residentes en los directorios /usr/bin y /usr/sbin.

/usr/share

Es un directorio para alojar datos y archivos de configuración independientes de la plataforma.

/usr/local

Tiene datos específicos para la plataforma, correspondientes a aplicaciones.

/var

Muchas veces se suele montar como un File System separado a partir de '/'. Contiene logs del sistema, spools de impresión, tareas scheduladas (a travñes del crontab,at) procesos en ejecución (/var/run), y demás recursos especiales del sistema.

/tmp

Mantiene archivos temporarios que se limpiarán cuando se reinicie el sistema. Existe además un /var/tmp con igual propósito, pero que tiene como diferencia la capacidad de mantener esos archivos a salvo aunque se reinicie el sistema.

/proc

9 El punto de montaje es el lugar del arbol de directorio a partir del cual se monta el file system. Ver a continuación el comando mount y la descripción del archivo fstab.

Es un pseudo file system que reside en memoria RAM generado por el sistema cuando arranca y mantenido a posteriori en tiempo real. Mantiene estados del kernel y de los procesos en archivos de texto planos fácilmente accesibles, y no siempre bien documentados.

El directorio /, es el origen del árbol de directorios y a partir de el podemos montar uno o mas file *systems*. También podemos montar un *file system* a partir de otro directorio de la estructura jerárquica que nace en /.

¿Que significa montar un *file system*?. En los sistemas UNIX los *file system* pueden ser locales, cuando residen en cualquiera de los discos del equipo que hostea el Sistema Operativo, o pueden residir en un equipo remoto al que el equipo local accede a través de una red de datos. En cualquier caso los *file system* se pueden montar (es decir hacerlos accesibles) o desmontar (es decir, volverlos invisibles a todos los usuarios, mediante dos comandos: *mount* y *umount*.

También actúan estos comandos (aunque en forma encubierta) cuando utilizamos una interfaz gráfica y conectamos un pen drive, u otro dispositivo con cualquier medio de almacenamiento. Sin embargo siempre es necesario recurrir a *mount* cuando se desea montar un *file system* remoto o cuando se desea visualizar el contenido de una partición del disco duro local que contiene por ejemplo un *file system* NTFS con una instalación windows.

El comando *mount* lleva dos argumentos: el dispositivo que controla al disco , DROM, DVD, pen drive, etc. que contiene el *file system*, y el punto de montaje que no es otra cosa que la ruta (Path) a partir de la cual se colgará ese *file system* completo del árbol jerárquico de directorios. Ej:

```
alejandro@notebook:~/info1$ mount /dev/scd0 /media/cdrom0/
mount: sólo el usuario root puede efectuar esta acción
alejandro@notebook:~/info1$
```

Primer detalle: solo el usuario *root* está habilitado para efectuar esta operación, lo cual es bastante lógico en virtud de lo que significa agregar un *file system* o bajarlo desde el punto de vista de la administración de los recursos del equipo y considerando que es una acción que afecta a los usuarios que están trabajando en el mismo. Por lo tanto es lógico que un usuario común no pueda por propia decisión bajar o levantar un *file system*.

De modo que nuevamente deberemos recurrir a los buenos oficios del comando *sudo*. Obviamente podemos iniciar una terminal como root si tenemos el password de este usuario. Esto es así con 100% de seguridad solo en nuestro equipo personal. No es práctica habitual (ni correcta) en un ámbito profesional donde integramos por ejemplo un equipo de desarrolladores de software. En esos casos tal vez ni tengamos habilitada la opción de sudo, ya que no estamos incluidos por el administrador del equipo en el archivo /etc/sudoers. Habrá que ver en cada situación. Hay ocasiones en las que hay

que recurrir al administrador sin otra salida posible para resolver un simple montaje. Así es la vida profesional: a veces tan rigurosa, que molesta pero es la forma de garantizar la integridad de un equipo en el que trabajan a veces cientos de personas.

```
alejandro@notebook:~/info1$ sudo mount /dev/scd0 /media/cdrom0/
[sudo] password for alejandro:
mount: dispositivo de bloques /dev/scd0 está protegido contra escritura; se monta como sólo lectura
alejandro@notebook:~/info1$
```

Vemos un mensaje que no debería llamar nuestra atención ya que lo que hemos montado es un DVD ROM.

```
alejandro@notebook:~/info1$ ls -las /media/cdrom0
total 10
2 dr-xr-xr-x 4 4294967295 4294967295  136 nov 12  2005 .
4 drwxr-xr-x 3 root       root       4096 feb 10 07:35 ..
2 dr-xr-xr-x 2 4294967295 4294967295   40 nov 12  2005 AUDIO_TS
2 dr-xr-xr-x 2 4294967295 4294967295  508 nov 12  2005 VIDEO_TS
alejandro@notebook:~/info1$
```

Por lo visto estamos listos para poder ver una película.

Para desmontarlo simplemente *umount* seguido del *path* a partir del cual se montó (y siempre como *root*):

```
alejandro@notebook:~/info1$ sudo umount /media/cdrom0
alejandro@notebook:~/info1$
```

Al desmontar no tenemos ninguna salida salvo en caso de que exista algún error.

```
alejandro@notebook:~/info1$ ls -las /media/cdrom0
total 8
4 drwxr-xr-x 2 root root 4096 dic 17  2008 .
4 drwxr-xr-x 3 root root 4096 feb 10 07:35 ..
alejandro@notebook:~/info1$
```

Sin dispositivo alguno montado el directorio de montaje como vemos queda vacío.

mount soporta además algunas opciones. La mas relevante es *-t*, que permite especificar el tipo de *file system* que se va a montar. Si no se especifica esta

opción *mount* trata de montar alguno de los *file systems* conocidos que se encuentran en */proc/filesystems*.

El sistema monta siempre dos file systems: / y /swap. Estos son vitales para su existencia. Cualquier otro file system que deseamos montar desde el inicio se debe declarar en un archivo del sistema especial para estos fines:

```
alejandro@notebook:~/info1$ cat /etc/fstab
# /etc/fstab: static file system information.
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
proc            /proc           proc    defaults          0       0
/dev/sda2       /               ext3    errors=remount-ro 0       1
/dev/sda5       none            swap    sw                0       0
/dev/scd0       /media/cdrom0   udf,iso9660 user,noauto   0       0
/dev/sda1       /mnt/windows    ntfs-3g defaults          0       0
alejandro@notebook:~/info1$
```

Como se puede apreciar este archivo de texto tiene información similar a la usada en el comando *mount*, y alguna otra información adicional. Lo interesante es que cada vez que el sistema arranca, para inicializar el *file system* lee este archivo y monta en las condiciones indicadas los dispositivos que allí se indican.

Las líneas que comienzan con '#' se asumen como comentarios.

En este caso solo se requiere conocer las particiones de la máquina y su contenido. Obsérvese la última línea: */dev/sda1* es la partición del disco duro que contiene la instalación de windows XP. Dicha partición está formateada en *NTFS*. La columna <type> tiene por función especificar lo mismo que escribiríamos a continuación de la opción *-t*. Como por razones complejas de comprender el kernel de Linux no monta en forma directa una partición *NTFS*, se utiliza un paquete adicional desarrollado por terceras partes bajo licencia GPL que se llama *ntfs-3G*.

Para ampliar la información *man fstab* tiene todo lo que se necesita conocer.

Procesos

Un proceso es (intentando ensayar una definición simplificada) una instancia de ejecución de un programa. El lector se podría preguntar cual es la diferencia. No es trivial. Tomemos como ejemplo el comando *ls*. Por un lado es un programa escrito en lenguaje C. Sin embargo siendo UNIX y LINUX sistemas multiusuario y pueden existir en un momento varias personas ejecutando *ls* en forma simultánea. Cada instancia de ejecución de *ls* es un proceso diferente.

El sistema operativo asigna a cada proceso una serie de atributos: prioridad (esto implica que porcentaje del tiempo de uso de la CPU tendrá asignada en relación al resto de los procesos que se ejecutan en el sistema), áreas de memoria, recursos, y otros parámetros que necesita cada sistema operativo para administrar su ejecución.

El comando ps sirve para visualizar el estado de los procesos.

```
alejandro@notebook:~/info1$ ps
  PID TTY          TIME CMD
 3888 pts/0    00:00:00 bash
 6479 pts/0    00:00:00 ps
alejandro@notebook:~/info1$
```

Provee información bastante escueta y solo de los procesos que ejecutan en la terminal. Sin embargo dá como para empezar a analizar algunos conceptos.

PID: Process ID. Es un número de 2 a 32 768 que Linux asigna a cada proceso. Lo selecciona secuencialmente. A medida que está activo el Sistema, los procesos que se van creando son asignados de un PID creciente. Al llegar a 32768, al siguiente proceso le asignará el número que haya quedado libre desde el 2 en adelante. Esto es de esperar ya que los procesos dinámicamente nacen y mueren. Por ejemplo cada vez que ejecutamos un ls para ver un directorio el sistema crea un proceso, al que le asigna un PID. EL proceso muestra el contenido del directorio en cuestión y finaliza quedando el PID vacante. Sin embargo como la asignación es la salida de un contador secuencial, ese número de PID no será reasignado sino hasta que el contador llegue a 32768 y regrese hasta ese valor.

TTY: Es la terminal asociada al proceso.

TIME: Horas:minutos:segundos de CPU acumuladas por el proceso.

CMD: comando que se ejecutó para crear el proceso.

En general para ver todos los procesos del sistema se agrega la opción *-e*. Queda a cargo del lector la comprobación.

Con la opción -f podemos listar mas información del proceso, como el User ID del dueño del proceso (el dueño es el usuario que ejecutó el comando para crear el proceso). PPID es el PID del proceso padre del proceso listado. C el uso del procesador (porcentual), y en CMD además del comando se presentan los argumentos inclusive.

Cuando un proceso nos trae problemas el comando que lo resuelve es *kill*. Lo veremos en el apartado siguiente ya que su uso efectivo necesita previamente de algunos elementos adicionales.

Mas acerca del shell....

El *shell* además de los comandos hasta ahora explicados (y muchísimos otros que se aprenden utilizando el sistema y teniendo un espíritu inquieto), posee operadores que le otorgan un gran poderío. Además existe algo llamado ENVIRONMET, que no es otra cosa que una colección de variables internas del sistema que se pueden modificar desde el *shell* y sirven para adoptar comportamientos default ante determinadas situaciones o para configurar la sesión que llevamos adelante.

UNIX fue pensado desde su inicio de manera tal de fomentar una alta reutilización de código.

Por ello es que el paradigma "everithing is a file" es un factor dominante en el diseño del sistema. Se tratan como archivos, a los dispositivos de E/S, lo cual simplifica dramáticamente su uso. Por ejemplo: para acceder a la placa de audio del equipo hay que manejar el "archivo"[10] */dev/dsp*. Escribiéndolo como un archivo común hacemos que el parlante de nuestro equipo reproduzca lo que le hemos escrito.

Esta concepción hace que con un grupo de 13 *system calls* resolvamos el acceso no solo a los archivos propiamente dichos sino a todo tipo de recursos. ¡Incluida la Entrada Salida!.

Por el contrario Windows posee un conjunto de funciones específicas para cada dispositivo, y todas diferentes. Esto hace que la programación sea extremadamente compleja y aumenta el riesgo de tener problemas ya que son muchas mas las rutinas involucradas.

Al trabajar con código reutilizable, resolvemos muchos mas problemas con las mismas pocas rutinas, minimizando la probabilidad de errores en el código.

En suma, cada vez que trabajemos con un archivo, o un dispositivo, o cualquier recurso que se pueda acceder con estas 13 *system calls*, obtendremos una referencia al mismo que se llama *file descriptor*. ¿Que es un *file descriptor*?. Simple: Un número entero positivo de 16 bits. Una vez obtenido se usa para todos los accesos que se necesite hacer al elemento referenciado por él.

Así cada proceso creado posee tres file descriptor por default ni bien es creado: *stdout* (acrónimo de *standard output*, que se refiere a la pantalla y cuyo valor es 1), *stdin* (acrónimo de *standard input*, que se refiere al teclado y cuyo valor es 0), y *stderr* (acrónimo de *standard error*, que generalmente se refiere a la pantalla y cuyo valor es 2). ¿Que significa esto?: La salida default de un proceso es la pantalla. Es decir, cada vez que deba enviar una salida, un resultado, un mensaje o lo que fuere al usuario, lo hará en la pantalla de la consola que tiene asignada el proceso. Esta consola no es otra que aquella

10 Por favor no perder de vista el concepto de dispositivo visto con anterioridad cuando listábamos el directorio /dev en bsca de elementos del file system que llevan en su primer columna el carácter 'c'. Tal es el caso de /dev/dsp.

desde cuyo *shell* se ejecutó el comando que derivó en el referido proceso. Lo mismo ocurre con los mensajes de error ya que la salida estándar de error es la pantalla. Y la entrada default del proceso (es decir, el dispositivo por el que el proceso esperará algún comando del usuario por default) es el teclado.

Uno de los operadores que introdujo el *shell* es '|', conocido como pipe. ¿De donde proviene ese nombre?. Una de las acepciones en inglés de pipe es tubo. Y así se comporta este comando.

Veamos como opera el siguiente comando:

```
ls -las /dev | more.
```

El directorio */dev* tiene muchas mas entradas de las que caben en la pantalla. El comando *ls -las /dev* realizará un scroll del contenido del directorio */dev* presentando en pantalla solo las últimas entradas.

El Comando *more* como ya vimos sirve para paginar un archivo de texto en la pantalla cuando este no puede ser mostrado íntegramente por cuestiones de extensión.

Por lo tanto si pudiésemos enviar la salida del comando *ls -las /dev* a la entrada del comando *more*, tendríamos resuelto el problema. Bien. Esto es lo que hace el operador '|'. Se comporta como un tubo (pipe en inglés) que une la salida de un comando con la entrada de otro.

Se pueden poner en cascada tantos como se quiera. Ejemplo:

```
ls -las /dev | grep ram | more
```

presentará en forma paginada todas las entradas del directorio */dev* en cuyo listado aparezca la palabra ram.

También resulta útil en ocasiones enviar a un archivo la salida de un proceso. Esto es fatalmente útil cuando nuestro compilador arroja muchísimos errores (mas de los que caben en una página. Espero que nunca el lector llegue a descubrir la utilidad de redirigir una larga lista de errores de compilación a un archivo, pero hay cosas que simplemente suceden aunque no queramos....

Resulta particularmente útil el uso de pipes con comandos que habitualmente arrojan salidas muy extensas como por ejemplo ps -ef. Supongamos que necesitamos conocer cuantas sesiones de shell hay en nuestro equipo:

```
alejandro@notebook:~/info1$ ps -ef | grep bash
500       3885  3626  0 23:21 pts/1    00:00:00 bash
500       3888  3623  0 23:22 pts/2    00:00:00 bash
500       3889  3623  0 23:22 pts/3    00:00:00 bash
500       3891  3623  0 23:22 pts/4    00:00:00 bash
500       3892  3623  0 23:22 pts/5    00:00:00 bash
```

```
500        3893  3623  0 23:22 pts/6    00:00:00 bash
500        3896  3623  0 23:22 pts/7    00:00:00 bash
500        3897  3623  0 23:22 pts/8    00:00:00 bash
500        3923  3623  0 23:22 pts/10   00:00:00 bash
500        3979  3623  0 23:22 pts/11   00:00:00 bash
500        3982  3623  0 23:22 pts/12   00:00:00 /bin/bash
500        3991  3623  0 23:22 pts/13   00:00:00 bash
500        3997  3623  0 23:22 pts/14   00:00:00 bash
500        4553  3668  0 23:22 pts/15   00:00:00 /bin/bash
root       4620  4610  0 23:23 pts/17   00:00:00 bash
500        5067  3885  0 23:42 pts/1    00:00:00 grep bash
alejandro@notebook:~/info1$
```

Sin el uso del filtro *grep* permite filtrar solo los procesos que nos interesan, que de otro modo saldrían y no en forma consecutiva listados entre todos los demás procesos.

Anteriormente nos referimos a un proceso que deseamos eliminar porque nos trae algún problema. O simplemente porque no necesitamos mas de su uso en el sistema, o porque no responde a ningún comando de control específico (es decir incluido en su propio código), o general (es decir los que dispone el Sistema Operativo para cualquier proceso). En este caso con el comando del listado anterior cambiando el argumento de *grep* es posible identificar a cualquier proceso, y actuar a posteriori.

```
alejandro@notebook:~/info1$ ps -ef | grep dumb
500        5572  3885  0 00:07 pts/1    00:00:00 dumb
500        5576  3885  0 00:07 pts/1    00:00:00 grep dumb
alejandro@notebook:~/info1$ kill 5572
alejandro@notebook:~/info1$ ps -ef | grep dumb
500        5576  3885  0 00:07 pts/1    00:00:00 grep dumb
alejandro@notebook:~/info1$
```

Como vemos es necesario conocer el process ID para luego terminar el proceso con el comando *kill*. A diferencia de otros casos de uso habitual, *kill* es inapelable. En el caso de "resistencia" o "rebeldías" por parte del proceso, la opción -9 es ..... insecticida.

¿Que pasa si volvemos a enviar el mismo comando kill al mismo processID?

```
alejandro@notebook:~/info1$ kill 5572
bash: kill: (5572) - No existe el proceso
```

```
alejandro@notebook:~/info1$
```

Es lógico, ya que según explicamos los procesos obtienen su número de un contador de 16 bits y hasta que no llegue al tope de la cuenta no volverá a pasar por los números mas bajos del rango del contador. De modo que volver a enviar el comando *kill* al mismo proceso no lo encontrará y generará esta salida diferente de la anterior.

En base a lo visto ¿puede el lector determinar que diferencia existe entre la salida del primer comando *kill* y la del segundo?. Obviamente la diferencia planteada va mas allá del texto del mensaje.

La respuesta es: en el primer caso la salida del comando se envió a *stdout*, y en el segundo caso (como en los subsiguientes si los hubiera, la salida se envía a *stderr*.

Para comprobarlo se sugiere c*orrer los siguientes comandos:*

```
alejandro@notebook:~/info1$ kill 6698 > out.txt 2>err.txt
alejandro@notebook:~/info1$ cat out.txt
alejandro@notebook:~/info1$ cat err.txt
alejandro@notebook:~/info1$ kill 6698 > out.txt 2>err.txt
alejandro@notebook:~/info1$ cat out.txt
alejandro@notebook:~/info1$ cat err.txt
bash: kill: (6698) - No existe el proceso
alejandro@notebook:~/info1$
```

En la primer línea se ejecuta el comando *kill* para matar al proceso 6698, enviado su salida (si la hubiere) al archivo *out.txt*. El resto de la línea de comando, redirige el file descriptor 2, es decir *stderr*, al archivo *err.txt*. Como vemos podemos separar las salidas estándar y de error a archivos diferentes, utilizando el operador redirección: '>'.

Cuando ejecutamos por segunda vez el mismo comando el proceso ya no existe y el mensaje que entrega *kill* y que habitualmente vemos en pantalla al intentar terminar un proceso que ya ha terminado, en realidad fue dirigido a *stderr*. El comando *cat* sobre el archivo *err.txt*, es la prueba de ello.

Si queremos que ambas salidas vayan al mismo archivo el operador '>&' nos permite redireccionar un file descriptor a otro.

Queda al lector comprobar que

```
alejandro@notebook:~/info1$ kill 6698 > kill.txt 2>&1
```

redirige la salida de *kill* y sus mensajes de error el mismo archivo, en este caso *kill.txt*.

Otra aplicación del operador de redirección '>' es cuando la salida de un comando es sumamente extensa. Por ejemplo:

```
alejandro@notebook:~/info1$ ls -las /dev > devices
alejandro@notebook:~/info1$ ls -las
total 28
 4 drwxr-xr-x  2 alejandro alejandro  4096 feb 10 17:10 .
 4 drwx------ 92 alejandro alejandro  4096 feb 10 07:47 ..
16 -rw-r--r--  1 alejandro alejandro 12775 feb 10 17:10 devices
 0 lrwxrwxrwx  1 alejandro alejandro     8 feb  7 05:35 passwd -> password
 4 -rw-r--r--  1 alejandro alejandro  1525 ene 24 12:24 password
alejandro@notebook:~/info1$
```

Como vemos no hubo salida por pantalla. Pero ahora nuestro directorio *info1* posee un archivo nuevo: *devices*.

Queda a cargo del lector ejecutar *more devices* para observar que el contenido de este archivo es la salida del comando *ls -las /dev*.

Cuando se usa el operador de redirección '>', se crea el archivo destino y si hay uno con el mismo nombre se trunca. Obsérvese que jamas el *shell* advertirá al usuario que hay un archivo con ese nombre. Lo trunca y nada mas. Silencioso e implacablemente obediente.

El operador redirección es bidireccional. Por ejemplo.

```
alejandro@notebook:~/info1$ ls -las
total 52
 4 drwxr-xr-x  2 alejandro alejandro  4096 feb 11 18:48 .
 4 drwx------ 92 alejandro alejandro  4096 feb 11 18:22 ..
16 -rw-r--r--  1 alejandro alejandro 12359 feb 10 17:20 devices
 8 -rwxr-xr-x  1 alejandro alejandro  6257 feb 10 17:55 dumb
 4 -rw-r--r--  1 alejandro alejandro    63 feb 10 17:55 dumb.c
 4 -rw-r--r--  1 alejandro alejandro    42 feb 11 00:55 err.txt
 4 -rw-r--r--  1 alejandro alejandro   702 feb 11 18:44 Nombres
 0 -rw-r--r--  1 alejandro alejandro     0 feb 11 00:55 out.txt
 0 lrwxrwxrwx  1 alejandro alejandro     8 feb  7 05:35 passwd -> password
 4 -rw-r--r--  1 alejandro alejandro  1525 ene 24 12:24 password
alejandro@notebook:~/info1$
```

En nuestro directorio info1 tenemos un archivo Nombres, que contiene a razón de uno por línea diferentes nombres escritos sin ningún tipo de ordenamiento.

Una forma de ordenarlo es:

```
alejandro@notebook:~/info1$ cat <Nombres | sort >NombresAZ
alejandro@notebook:~/info1$
```

Observar el comportamiento de cat sin argumentos, es decir tipear simplemente *cat* seguido de enter. Por cada línea que escribimos la presenta una vez tipeado enter en el renglón siguiente. Es decir que sin un archivo como argumento, espera sus comandos desde *stdin*. Con el operador '<' le hemos indicado que el archivo Nombres se convierte en *stdin*.

El lector puede comprobar el ordenamiento de la salida *NombresAZ* mediante el comando *cat NombresAZ.*

Si lo que desea es ordenarlo en forma alfabética pero decreciente, puede hacerlo de dos formas:

Hacer uso de la opción -r del comando sort:

```
alejandro@notebook:~/info1$ cat <Nombres | sort -r >NombresZA
alejandro@notebook:~/info1$
```

O bien tomar el resultado del ordenamiento creciente y mediante el conado tac (inversa literal de cat) imprimirlo redirección mediante a otro archivo.

```
alejandro@notebook:~/info1$ tac NombresAZ > NombresZA
alejandro@notebook:~/info1$
```

Si deseamos anexar información a un archivo existente el operador es '>>'. En este caso. Si el archivo no existe lo crea y si existe le agrega la nueva información a continuación de la existente.

Para comprobarlo ejecutemos el siguiente comando:

```
alejandro@notebook:~/info1$ echo $HOME >> devices
```

Ejecute a continuación

```
alejandro@notebook:~/info1$ cat devices
```

y observe la última línea del archivo. Esa línea es el resultado de *echo $HOME*.

¿Que es *$HOME*?. Vamos por partes: *HOME* es una variable del *ENVIRONMET* que contiene el *path* del *home directory* del usuario. El operador '$' sirve para acceder al contenido de una variable de entorno desde el *shell*. A continuación vemos la diferencia de usar y no usar '$' al referirse a *HOME*

```
alejandro@notebook:~/info1$ echo HOME
HOME
alejandro@notebook:~/info1$ echo $HOME
/home/alejandro
```

```
alejandro@notebook:~/info1$
```

En el primer caso tomo a *HOME* como una simple cadena de caracteres y en el segundo el operador '$' permitió acceder al contenido de la variable de entorno (*ENVIRONMET* es entorno) llamada *HOME*.

El comando *echo* es un simple utilitario que sirve para mostrar en *stdout* lo que le pasemos como argumento. Mas adelante veremos su utilidad en algunos ejemplos mas elaborados.

Volviendo al *environment* o entrono, nos referimos a él como un conjunto de variables internas. ¿Podemos verlas?. Si podemos. El comando es *printenv*, o simplemente *env*. Sin argumentos presenta todas las variables. Con el nombre de una variable como argumento, solo nos muestra el valor de la misma.

Debido a la longitud de la salida queda a cargo del lector ejecutar *printenv* sin argumentos en su equipo. Además para ampliar información *man environ*, provee muy buena información

Veamos el siguiente ejemplo:

```
alejandro@notebook:~/info1$ printenv PATH
  /home/alejandro/bin:/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/local/cu
da/bin:.:/usr/local/cuda/bin:.
alejandro@notebook:~/info1$
```

Otra forma de acceder al valor de una variable de entorno, es mediante el comando *echo*.

```
alejandro@notebook:~/info1$ echo $PATH
  /home/alejandro/bin:/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/local/cu
da/bin:.:/usr/local/cuda/bin:.
alejandro@notebook:~/info1$
```

La variable de entorno *PATH* fue creada en UNIX para escribir en ella las rutas en las que el *shell* buscará al comando requerido por el usuario si no se provee en el mismo el *path absoluto*. ¿Que es el *path absoluto*?. Es la ruta (*path*) completa del archivo desde el directorio raíz hasta el subdirectorio en el que se encuentra. Por ejemplo */bin/bash*, es el *path* absoluto de nuestro *shell*.

La verdad es que nunca recordamos la ruta completa de un ejecutable así que escribimos su nombre "a secas". La variable *PATH* tiene entonces las rutas de los comandos mas comunes del sistema. El *shell* buscará en esas rutas y si no lo encuentra arrojará el mensaje correspondiente pos *stderr*.

```
alejandro@notebook:~/info1$ comando
bash: comando: command not found
alejandro@notebook:~/info1$
```

En nuestro ejemplo (un tanto burdo por cierto) *comando*, no existe con ese nombre. Sin embargo el *shell* lo buscó solo en los directorios especificados en el variable *PATH* impresa en el otro listado.

PATH se compone de rutas separadas por el carácter ':'.

El programa de instalación del Sistema Operativo inicializa las variables de entorno con los valores adecuados al momento de la instalación. Una vez en operación los valores iniciales almacenados en las variables de entorno pueden modificarse de acuerdo con las necesidades que van apareciendo.

En el archivo *~/.bashrc* se pueden volcar las modificaciones para que sean tomadas de modo permanente cada vez que nuestro usuario inicie sesión.

Incluso podemos crear una variable de *environmet* propia. Por ejemplo, en el siguiente ejemplo nos creamos una variable *WORK*.

```
alejandro@notebook:~/info1$ echo $HOME
/home/alejandro
alejandro@notebook:~/info1$ WORK=$HOME/work
alejandro@notebook:~/info1$ cd $WORK
alejandro@notebook:~/work$
```

Si nos resulta útil la podemos incluir en el archivo *~/.bashrc*, para que se cree cada vez que nuestro usuario inicie sesión.

Muchas veces al instalar una aplicación nos informa que ha modificado una variable de entorno (típicamente *PATH*). Esto se logra mediante las *system calls getenv ()* y *setenv ()*, que leen y escriben variables de entorno respectivamente.

El shell como lenguaje de programación (solo un vistazo)

En este punto es conveniente advertir al lector que si aun no posee experiencia con algún lenguaje de programación conviene posponer la lectura de este capítulo hasta tener alguna gimnasia de programación en lenguaje C.

Si en cambio ya hay experiencia en el uso de algún lenguaje (aunque no sea el C) es posible continuar leyendo.

El *shell* como ya hemos dicho es mucho mas que un intérprete de comandos. Es en rigor un lenguaje de programación en si mismo. Veámoslo a través de un ejemplo.

Supongamos que necesitamos copiar a nuestro directorio info1, una serie de archivos de programas escritos en C, pero solo los que invocan a la función *pipe*. Los programas están a partir del *path*:

/home/alejandro/work/facu/TDIII/Programming/FullTPs/ejemplos

y encima los archivos se encuentran agrupados en diferentes subdirectorios a partir de ese path!!

No es simple hacerlo a mano y no hay un comando que pueda resolverlo por si solo.

Por lo tanto la solución pasa por utilizar recursos del shell que aún no conocemos, pero que como veremos son de gran utilidad y versatilidad.

```
alejandro@notebook:~/info1$ for dir in `ls -d
/home/alejandro/work/facu/TDIII/Programming/FullTPs/ejemplos/*`
> do
> cd $dir
> for file in `ls`
> do
> if grep -l pipe $file
> then cp $file ~/info1
> fi
> done
> cd ~/info1
> done
alejandro@notebook:~/info1$
```

Las sentencias de control de flujo *for* e *if* del ejemplo (así como *while* que en este caso no se ha utilizado) son parte del *shell*. Para comprobarlo *man bash* (o *man* del *shell* que el lector esté utilizando, *tcsh*, *csh*, o *sh*) y comprobará la potencia de este programa.

Cada vez que dejamos algo pendiente el *prompt* default del *shell* ('$', o '#' si es *root*), cambia por '>' para indicar que se debe completar.

En este listado existen varios conceptos, además de la potente capacidad de control de flujo que dispone el *shell*.

Una característica del shell puesta en juego en este complejo comando es el encerrar entre ` ` a los comandos. Cuando eso ocurre, el resultado del comando se asigna a la sentencia.

De modo que al decir

```
for file in `ls`
```

la variable *file* se carga con cada una de las líneas de salida del comando *ls* a razón de una por cada ciclo *for*.

Lo mismo ocurre en otros puntos del comando en los que utilizamos esta facilidad.

Básicamente por cada entrada obtenida por *ls -d* (la cual se asume como un directorio) ingresamos al mismo y copiamos cada archivo que contiene la cadena "pipe" al directorio destino.

Otra interesante opción para observar es -l en el comando grep. De este modo en lugar de escribir a stdout las líneas del archivo que contienen la cadena buscada, escribe solo el nombre del archivo.

Otra forma de escribirlo (mas compleja aún y solo apta para usuarios mas avanzados) es

```
alejandro@notebook:~/info1$ for dir in `ls -d
/home/alejandro/work/facu/TDIII/Programming/FullTPs/ejemplos/*`;
do cd $dir; for file in `ls`; do if grep -l pipe $file; then cp
$file ~/info1; fi ; done ; cd ~/info1 ; done
```

Es decir cada línea separada por un carácter ';'.

Una forma mas razonable de utilizar estas facilidades del *shell*, es utilizando archivos de *script*. Estos archivos son archivos planos de texto que contienen comandos del *shell* y que el *shell* puede interpretar de modo de ejecutar línea por línea en forma secuencial, permitiendo construir comandos mas complejos reutilizando los comandos disponibles, combinándolos adecuadamente y haciendo uso de las capacidades de control de flujo del shell. Para ello simplemente hay que darle al archivo de texto permisos de ejecución.

```
# ! /bin/bash
# El carácter '#' indica que lo que sigue es un  comentario.
# El shell ignora estas líneas. Podemos utilizarlas para explicar que hace
# nuestro script
# Este script ingresa al primer nivel de subdirectorios a partir del path
# que se pasa como argumento y copia al directorio ~/info1 todos los
# archivos de programas C que contengan la palabra que se recibe como
# segundo argumento.
for dir in `ls -d $1`
do
      cd $dir
      for file in `ls`
            do
                  if grep -l $2 $file
                  then
                        cp $file ~/info1
                  fi
            done
```

```
      cd ~/info1
done
```

Vemos que los datos que por línea de comando ingresamos en forma "dura" aquí los leemos desde variables cuyo nombre corresponde al número de orden que tienen en la línea de comando. $1 accede al contenido del 1er. argumento y $2 al segundo. Si hubiese mas argumentos la secuencia continúa.

Salvando el archivo y dándole permisos de ejecución con chmod, queda resuelto el uso de nuestro comando.

Podemos utilizar editores complejos como el vi o mas simples como el nano.

No obstante una vez terminado el archivo y salvado con el editor, nos queda en este estado

```
alejandro@notebook:~/info1$ ls -las
total 112
 4 drwxr-xr-x  2 alejandro alejandro  4096 feb 13 23:33 .
 4 drwx------ 92 alejandro alejandro  4096 feb 13 20:17 ..
16 -rw-r--r--  1 alejandro alejandro 12359 feb 10 17:20 devices
 8 -rwxr-xr-x  1 alejandro alejandro  6257 feb 10 17:55 dumb
 4 -rw-r--r--  1 alejandro alejandro    63 feb 10 17:55 dumb.c
 4 -rw-r--r--  1 alejandro alejandro    42 feb 11 00:55 err.txt
 4 -rw-r--r--  1 alejandro alejandro   702 feb 11 18:44 Nombres
 4 -rw-r--r--  1 alejandro alejandro   702 feb 11 18:53 NombresAZ
 4 -rw-r--r--  1 alejandro alejandro   702 feb 11 21:08 NombresZA
 0 -rw-r--r--  1 alejandro alejandro     0 feb 11 00:55 out.txt
 0 lrwxrwxrwx  1 alejandro alejandro     8 feb  7 05:35 passwd -> password
 4 -rw-r--r--  1 alejandro alejandro  1525 ene 24 12:24 password
 4 -rw-r--r--  1 alejandro alejandro   552 feb 13 23:33 rcopywf
alejandro@notebook:~/info1$
```

el archivo *rcopywf* (recursive copy with filter, así lo llamamos), no tiene permisos de ejecución. Por lo tanto para resolverlo hacemos:

```
alejandro@notebook:~/info1$ chmod a+x rcopywf
alejandro@notebook:~/info1$ ls -las
total 112
 4 drwxr-xr-x  2 alejandro alejandro  4096 feb 13 23:33 .
 4 drwx------ 92 alejandro alejandro  4096 feb 13 20:17 ..
16 -rw-r--r--  1 alejandro alejandro 12359 feb 10 17:20 devices
 8 -rwxr-xr-x  1 alejandro alejandro  6257 feb 10 17:55 dumb
```

```
 4 -rw-r--r--  1 alejandro alejandro    63 feb 10 17:55 dumb.c
 4 -rw-r--r--  1 alejandro alejandro    42 feb 11 00:55 err.txt
 4 -rw-r--r--  1 alejandro alejandro   702 feb 11 18:44 Nombres
 4 -rw-r--r--  1 alejandro alejandro   702 feb 11 18:53 NombresAZ
 4 -rw-r--r--  1 alejandro alejandro   702 feb 11 21:08 NombresZA
 0 -rw-r--r--  1 alejandro alejandro     0 feb 11 00:55 out.txt
 0 lrwxrwxrwx  1 alejandro alejandro     8 feb  7 05:35 passwd -> password
 4 -rw-r--r--  1 alejandro alejandro  1525 ene 24 12:24 password
 4 -rwxr-xr-x  1 alejandro alejandro   552 feb 13 23:33 rcopywf
```

Como vemos le hemos agregado permisos de ejecución para el dueño, los usuarios del mismo grupo del dueño y para el resto. Por eso utilizamos la opción *a+x*, de modo que asigne permiso de ejecución (*x*) para todos (*a* de all) los usuarios.

Cualquier programa que utiliza argumentos por línea de comandos y es medianamente decente chequea que lleguen los argumentos necesarios (al menos la cantidad de estos)

Agregando estas líneas justo al inicio del script se controla que a menos se tipeen dos argumentos.

```
if [ "$2" == "" ]
then
        echo "Número incorrecto de argumentos"
        echo "Uso: rcopywf path_origen string_contenido"
        exit
fi
```

Si $2 es nulo, significa que no hay dos argumentos. Por lo tanto el script no debe ejecutar.

Conclusiones.

Hemos recorrido un vistazo rápido sobre un sistema operativo, cuyo fin es comenzar a andar.

¿Queda mucho camino aun?. Si. Mucho. Pero se hace con el uso, y la consulta de las man pages, y la documentación disponible en internet.

El fin de esta primer guía es solo dar el impulso inicial suficiente como para arrancar.