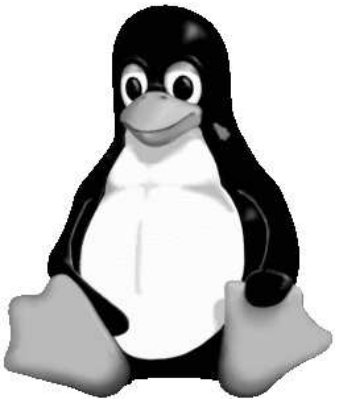
Linux: Primeros Pasos en la FCAGLP



Versión 0.2-1e, 19 de abril de 2002.
Fecha de impresión: 19 de abril de 2002

Índice General

Prefacio

Esta es una guía para nuevos usuarios del sistema Linux, dirigida a los alumnos de la Facultad de Ciencias Astronómicas y Geofísicas principiantes en UNIX. Hemos pretendido ser tan genéricos como nos ha sido posible de tal modo que pueda ser aplicable a cualquiera de las distribuciones de software para Linux.

Tenemos intenciones de extender esta guía a otros temas relacionados con el Linux y UNIX en general; es por esto que se presenta con un **Capítulo 1** y no hay nigún otro :-); esperamos poder agregar otros capítulos en cuanto nos sea posible. Los Alumnos de la Cátedra deberán ser quienes lo evalúen, critiquen y sugieran qué otro tipo de temas les gustaría que sean incluídos. Por el momento sólo se nos ha ocurrido agregar un capítulo dedicado a cubrir, en lo que podamos, el lenguaje FORTRAN.

Federico Bareilles
8 de Septiembre de 1998

[1] `mdw@sunsite.unc.edu`

Capítulo 1

Guía de Linux

Autores: Matt Welsh[1], Larry Greenfield y Karl Fogel
Traducción: Eduardo Lluna Gil[2]
Adaptación: Federico Bareilles[3]

1.1 Introducción

Los nuevos usuarios de UNIX y Linux pueden estar un poco intimidados por el tamaño y aparente complejidad del sistema que tienen ante sí. Hay muchos buenos libros sobre el uso de UNIX para todos los niveles, desde novatos a expertos. Pero ninguno de estos libros cubre específicamente una introducción al uso de Linux. Mientras que el 95% del uso de Linux es exactamente como cualquier otro UNIX, la forma más clara de comenzar con su nuevo sistema es una guía a medida para Linux. He aquí esa guía.

Este capítulo no presentará gran cantidad de detalles ni cubrirá temas muy avanzados. Sino que está pensado para permitir al nuevo usuario de Linux comenzar a usar el sistema y situarlo en una posición en la que él o ella puedan leer libros más generales sobre UNIX y entender las diferencias básicas entre otros sistemas UNIX y Linux.

Se va a presuponer muy poco, excepto quizá alguna familiaridad con las computadoras personales y MS-DOS. Pero incluso si no es un usuario de MS-DOS, debería ser capaz de entender cualquier cosa de la que hablemos. A primera vista, UNIX parece ser como MS-DOS (después de todo, partes de MS-DOS fueron tomadas de CP/M, el cual fue a su vez inspirado en UNIX). Pero sólo las características superficiales de UNIX se parecen a MS-DOS. Incluso si usted es completamente nuevo en el mundo de las PC, esta guía debería serle de ayuda.

Y, antes de comenzar: *No tenga miedo de experimentar.* El sistema no lo morderá. No puede destruir nada trabajando con él. UNIX tiene ciertos sistemas de seguridad para evitar que usuarios "normales" (del tipo que suponemos que es usted) dañen archivos esenciales para el sistema. En el peor de los casos que es que borre todos sus archivos, sólo afectará al usuario que lo haya hecho y no al sistema u otros usuarios.

[1] mdw@sunsite.unc.edu
[2] elluna@aii.upv.es
[3] fede@iar.unlp.edu.ar

1.2 Conceptos básicos de UNIX

UNIX es un sistema operativo multitarea y multiusuario. Esto significa que puede haber más de una persona usando una computadora a la vez, cada una de ellas ejecutando a su vez diferentes aplicaciones. (Esto difiere de MS-DOS, donde sólo una persona puede usar el sistema en un momento dado). Bajo UNIX, para que los usuarios puedan identificarse en el sistema, deben **presentarse (log in)**, proceso que consta de dos pasos: Introducir el **nombre de usuario (login o username)** (el nombre con que será identificado por el sistema), y una **palabra de acceso o ingreso (password)**, la cual es su llave personal secreta o contraseña para entrar en la cuenta. Como sólo usted conoce su palabra de acceso (password desde ahora), nadie más podrá presentarse en el sistema con su nombre de usuario.

En los sistemas UNIX tradicionales, el administrador de éste asignará el nombre de usuario y una password (palabra de acceso) inicial en el momento de crear la cuenta de usuario. Para el resto de las discusiones, usaremos el nombre de usuario "`nahuel`".

Además, cada sistema UNIX tiene un **nombre del sistema (hostname)** asignado. Este "hostname" le da nombre a la máquina, además de carácter y encanto. El nombre del sistema es usado para identificar máquinas en una red, pero incluso aunque la máquina no esté en red, debería tener su nombre. En nuestros ejemplos, el nombre del sistema (o computadora) será "`arrakis`"

1.2.1 Presentación en el sistema (loggin in)

En el momento de presentarse en el sistema, verá la siguiente línea de comandos en la pantalla:

```
Bienvenido a Linux 2.0.32.

Recuerde que NO se debe FUMAR en la sala de Computacion

arrakis login:
```

Ahora, introduzca su nombre de usuario y pulse `Return`. Nuestro héroe `nahuel`, teclearía lo siguiente:

```
arrakis login:  nahuel
Password:
```

Ahora introduzca su password. Esta no será mostrada en la pantalla mientras se la va tecleando, por lo que debe teclear cuidadosamente (evitando equivocarse). Si introduce una password incorrecta, se mostrará el siguiente mensaje:

```
Login incorrect
```

y deberá intentarlo de nuevo.

Una vez que ha introducido correctamente el nombre de usuario y la password, está oficialmente "presentado" en el sistema y listo para comenzar a trabajar.

1.2.2 Consolas virtuales

La **consola** del sistema está constituida por el monitor y el teclado, conectados directamente (fisicamente) a éste. Como UNIX es un sistema operativo multiusuario,

puede tener otras terminales conectadas a puertos serie del sistema (como la terminal que conectamos con el "`telix`" en el MS-DOS), pero éstos no serán la consola). Linux, como otras versiones de UNIX, porporciona acceso a **consolas virtuales** (o VC's), las cuales le permitirán tener más de una sesión de trabajo activa desde la consola a la vez.

Para demostrar esto, entre en su sistema (como hemos visto antes). Ahora pulse `alt-F2`. Debería ver la pregunta `login:` de nuevo. Está viendo la segunda consola virtual—ha entrado en el sistema por la primera—. Para volver a la primera VC, pulse `alt-F1`. *Voila!* ha vuelto a la primera sesión.

Un sistema Linux recién instalado probablemente le permita acceder a las primeras seis VC's, usando `alt-F1` a `alt-F6`. Pero es posible habilitar hasta 12 VC's; una por cada tecla de función del teclado. Como puede ver, el uso de VC's es muy potente: puede estar trabajando en diferentes VC's a la vez. En general, la consola en `alt-F7` y superiores, corresponde al entorno gráfico.

Aunque el uso de VC's es algo limitado (después de todo, sólo puede mirar un VC cada vez), esto debería darle una idea de las capacidades multiusuario del sistema. Mientras está trabajando en el VC #1, puede conmutar al VC #2 y comenzar a trabajar en otra cosa; mientras que el programa que corre en la VC #1 ¡continúa su tarea!

1.2.3 Intérpretes de comandos (shell) y comandos

En la mayoría de las exploraciones en el mundo de UNIX, estará hablando con el sistema a través del uso de un **intérprete de comandos**, conocido como "shell". Un intérprete de comandos es simplemente un programa que toma la entrada del usuario (p.ej. las órdenes que teclea) y las traduce a instrucciones. Esto puede ser comparado con el `COMMAND.COM` de MS-DOS, el cual efectúa esencialmente la misma tarea. El intérprete de comandos es sólo una de las interfaces con UNIX. Hay muchas interfaces posibles— como el sistema X Windows, el cual le permite ejecutar comandos usando el ratón y el teclado en un entorno gráfico.

Tan pronto como entra en el sistema, éste arranca un intérprete de comandos y Ud. ya puede teclear órdenes. Veamos un ejemplo rápido. Aquí, Nahuel entra en el sistema y es situado en el intérprete de comandos

```
arrakis login:  nahuel
Password:  nahuel's password
Welcome to arrakis!

arrakis:~$
```

" `arrakis:~$` " es el "prompt" del intérprete de comandos [4], indicando que está listo para recibir órdenes. Tratemos de decirle al sistema que haga algo interesante:

```
arrakis:~$ make love
make:  *** No rule to make target 'love'.  Stop.
arrakis:~$
```

Bien, como resulta que `make` es el nombre de un programa ya existente en el sistema, el intérprete de comandos lo ejecuta. (Desafortunadamente, el sistema no está siendo muy amigable).

[4] Éste puede variar de sistema en sistema; incluso cada usuario puede personalizarlo

Esto nos lleva a una cuestión importante: ¿Qué son las órdenes? ¿Qué ocurre cuando tecleamos "`make love`"? La primera palabra de la orden, "`make`", es el nombre de la orden a ejecutar. El resto de la orden es tomado como argumento de la orden. Ejemplos:

```
arrakis:~$ cp foo bar
```

Aquí, el nombre de la orden es "`cp`", y los argumentos son "`foo`" y "`bar`".

Cuando teclea una orden, el intérprete de comandos hace varias cosas. Antes que nada, busca el nombre de la orden y comprueba si es una orden interna. (Es decir, una orden que el intérprete de comandos sabe ejecutar por sí mismo; más adelante veremos varias órdenes de ese tipo). El intérprete de comandos también comprueba si la orden es un "alias" o nombre sustituto de otra orden. Si no se cumple ninguno de estos casos, el intérprete de comandos busca el programa y lo ejecuta pasándole los argumentos especificados en la línea de comandos.

En nuestro ejemplo, el intérprete de comandos busca el programa llamado `make` y lo ejecuta con el argumento `love`. `make` es un programa usado a menudo para compilar programas grandes, y toma como argumentos el nombre de un "objetivo" a compilar. En el caso de "`make love`", ordenamos a `make` que compile el objetivo `love`. Como `make` no puede encontrar un objetivo de ese nombre, falla enviando un mensaje de error y volviendo al intérprete de comandos.

¿Qué ocurre si tecleamos una orden y el intérprete de comandos no puede encontrar el programa de ese nombre?. Bien, probémoslo:

```
arrakis:~$ eat dirt
bash:  eat:  command not found
arrakis:~$
```

Bastante simple, si no se puede encontrar el programa con el nombre dado en la orden (aquí "`eat`"), se muestra un mensaje de error que debería de ser autoexplicativo. A menudo verá este mensaje de error si se equivoca al teclear una orden (por ejemplo, si huviese tecleado "`mkae love`" en lugar de "`make love`".

1.2.4 Salida del sistema

Antes de ahondar más, deberíamos ver cómo salir del sistema. Desde la línea de órdenes usaremos el comando:

```
arrakis:~$ exit
```

para salir. Hay otras formas, pero ésta es la más fácil.

1.2.5 Cambiando la palabra de acceso (password)

También debe asegurarse de la forma de cambiar su password. La orden `passwd` le pedirá su password vieja y la nueva. Volverá a pedir una segunda vez la nueva para confirmarla. Tenga cuidado de no olvidar su password; si eso ocurre, deberá pedirle al administrador del sistema que la modifique por usted.

1.2.6 Archivos y directorios

Bajo la mayoría de los sistemas operativos (UNIX incluido), existe el concepto de **archivo** (fichero, en algunas traducciones), el cual es un conjunto de información al que se le ha asignado un nombre (llamado **nombre del archivo**). Ejemplos de archivo son un mensaje de correo, o un programa que puede ser ejecutado. Esencialmente, cualquier cosa grabada en el disco de la computadora es guardada en un archivo individual.

Los archivos son identificados por sus nombres. Por ejemplo, el archivo que contiene su historial podría ser grabado con el nombre `history-paper`. Estos nombres usualmente identifican el archivo y su contenido de alguna forma significativa para usted. No hay un formato estándar para los nombres de los archivos como lo hay en MS-DOS y en otros sistemas operativos; en general estos pueden contener cualquier caracter (excepto `/` — ver la discusión sobre "pathnames" (rutas de archivos) más adelante), y están limitados a 256 caracteres de longitud.

Con el concepto de archivo aparece el concepto de directorio. Un **directorio** es simplemente una colección de archivos. Puede ser considerado como una "carpeta" (folder) que contiene muchos archivos diferentes. Los directorios también tienen nombre con el que los podemos identificar. Además, los directorios mantienen una estructura de árbol, es decir, éstos pueden contener otros directorios.

Un archivo puede ser referenciado por su **nombre con camino**, el cual está constituido por su nombre, antecedido por el nombre del directorio que lo contiene. Por ejemplo, supongamos que Nahuel tiene un directorio de nombre `papers` que contiene tres archivos: `history-final`, `english-lit` y `masters-thesis`. (Cada uno de los tres archivos contiene información sobre tres de los proyectos en los que Nahuel está trabajando). Para referirse al archivo `english-lit`, Nahuel puede especificar su camino:

```
papers/english-lit
```

Como puede ver, el directorio y el nombre del archivo van separados por un caracter "`/`". Por esta razón, los nombres de archivo no pueden contener este caracter. Los usuarios de MS-DOS encontrarán familiar esta convención, aunque en el mundo MS-DOS se usa el caracter "\". Digamos que, en MS-DOS, es al revés.

Como hemos mencionado, los directorios pueden anidarse uno dentro de otro. Por ejemplo, supongamos que Nahuel tiene otro directorio dentro de `papers` llamado `netes` y alli el archivo `cheat-sheet`. El camino de este archivo sería

```
papers/notes/cheat-sheet
```

Por lo tanto, el camino realmente es la "ruta" que se debe tomar para localizar un archivo. El directorio sobre un subdirectorio dado es conocido como el **directorio padre**. Aquí, el directorio `papers` es el padre del directorio `notes`.

1.2.7 El árbol de directorios

La mayoría de los sistemas UNIX tienen una distribución de archivos estándar, de forma que recursos y archivos puedan ser fácilmente localizados. Esta distribución forma el árbol de directorios, el cual comienza en el directorio "`/`", también conocido como "directorio raíz" (root directory). Directamente por debajo de `/` hay algunos subdirectorios importantes: `/bin`, `/etc`, `/dev` y `/usr`, entre otros. Éstos a su vez

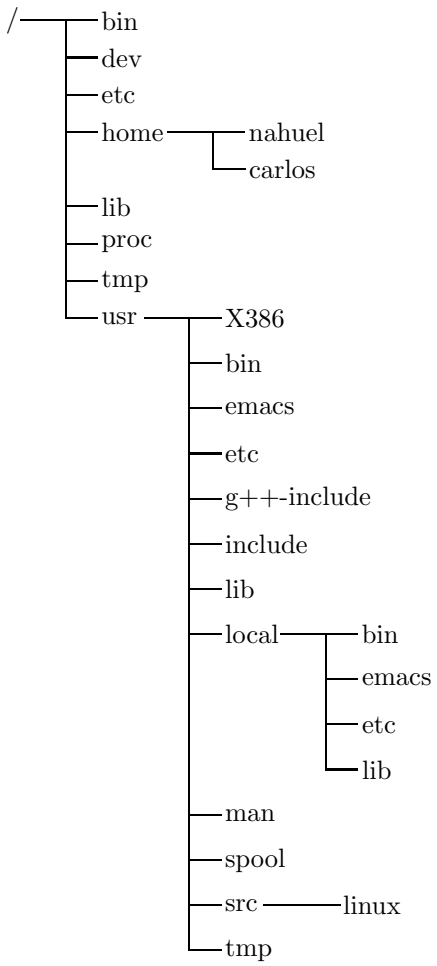


Figura 1.1: Típico árbol de directorios Unix (resumido).

contienen otros directorios con archivos de configuración del sistema, programas, etc.

En particular, cada usuario tiene un **directorio "home"**. Este es el directorio en el que el usuario guardará sus archivos. En los ejemplos anteriores, todos los archivos de Nahuel (como `cheat-sheet` y `history-final`) estaban contenidos en el directorio "home" de Nahuel. Usualmente, los directorios home de los usuarios cuelgan de `/home` y son nombrados con el nombre del usuario al que pertenecen. Por lo tanto, el directorio "home" de Nahuel es `/home/nahuel`.

En la Figura 1.1 (página 6) se muestra un árbol de directorio de ejemplo. Este debería darle una idea de como está organizado en su sistema el árbol de directorios.

1.2.8 Directorio de trabajo actual

En cualquier momento, las órdenes que teclee al intérprete de comandos son dadas en términos de su **directorio de trabajo actual**. Puede pensar en su directorio actual de trabajo como en el directorio en el que actualmente está "situado". Cuando

entra en el sistema, su directorio de trabajo se inicializa a su directorio home; `/home/nahuel` en nuestro caso. En cualquier momento que referencie a un archivo, puede hacerlo en relación a su directorio de trabajo actual, en lugar de especificar el camino completo del archivo.

Veamos un ejemplo. Nahuel tiene el directorio `papers`, y `papers` contiene el archivo `history-final`. Si Nahuel quiere echar un vistazo a ese archivo, puede usar la orden (para salir tipee `q`).

```
arrakis:~$ more /home/nahuel/papers/history-final
```

La orden `more` simplemente muestra el archivo, pantalla a pantalla. Pero, como el directorio de trabajo actual de Nahuel es `/home/nahuel`, podría haberse referido al archivo de forma *relativa* a su directorio de trabajo actual. La orden sería

```
arrakis:~$ more papers/history-final
```

Por lo tanto, si comienza el nombre de un archivo (como `papers/final`) con un carácter distinto a "`/`", el sistema supone que se está refiriendo al archivo con su posición relativa a su directorio de trabajo. Esto es conocido como **camino relativo** (**relative path**).

Por otra parte, si comienza el nombre del archivo con "`/`", el sistema interpreta esto como un camino completo—es decir, el camino al archivo completo desde el directorio raíz, `/`. Esto es conocido como **camino absoluto** (**absolute path**).

1.2.9 Refiriéndose al directorio "home"

Bajo `tcsh` y `bash`,[5] el directorio "home" puede ser referenciado usando el caracter de la tilde (`~`). Por ejemplo, la orden

```
arrakis:~$ more ~/papers/history-final
```

es equivalente a

```
arrakis:~$ more /home/nahuel/papers/history-final
```

El caracter "`~`" es simplemente sustituido por el intérprete de comandos, con el nombre del directorio "home".

Además, también puede especificar otros directorios "home" de usuarios con la tilde. El camino "`~carlos/letters`" es traducido por el intérprete de órdenes a "`/home/carlos/letters`" (si `/home/carlos` es el directorio "home" de Carlos). El uso de la tilde es simplemente un atajo; no existe ningún directorio llamado "`~`"—es (simplemente) una ayuda sintáctica proporcionada por el intérprete de comandos.

1.3 Primeros pasos en UNIX

Antes de comenzar es importante destacar que todos los nombres de archivos y comandos hacen diferencia entre mayúsculas y minúsculasson (case-sensitive) , a diferencia de sistemas operativos como MS-DOS. Por ejemplo, el comando `make` es diferente a `Make` o `MAKE`. Lo mismo ocurre en el caso de nombres de archivos o directorios.

[5] `tcsh` y `bash` son dos *intérpretes de comandos* que corren bajo Linux. Un intérprete de comandos es el programa que lee las órdenes del usuario y las ejecuta; la mayoría de los sistemas Linux habilitan `tcsh` o `bash` para las nuevas cuentas de usuario.

1.3.1 Moviéndonos por el entorno

Ahora que ya podemos presentarnos como usuarios, y sabemos cómo indicar archivos con su camino completo, ¿cómo podemos cambiar nuestro directorio de trabajo?

La orden para movernos por la estructura de directorios es `cd`, abreviación de "cambio de directorio". Hay que destacar que la mayoría de las órdenes Unix más usadas son de dos o tres letras. La forma de uso de la orden `cd` es:

`cd` ⟨*directorio*⟩

donde ⟨*directorio*⟩ es el nombre del directorio al que queremos ir.

Como dijimos, al entrar al sistema comenzamos en el directorio "home". Si Nahuel quiere ir al subdirectorio `papers`, debería usar la orden

```
arrakis:~$ cd papers
arrakis:~/papers$
```

Como se puede ver, la línea de comandos de Nahuel cambia para mostrar su directorio actual de trabajo. Ahora que ya está en el directorio `papers` puede echarle un vistazo a su archivo `history-final` con el comando

```
arrakis:~/papers$ more history-final
```

Ahora Nahuel está en el subdirectorio `papers`, para volver al directorio padre de éste, usará la orden

```
arrakis:~/papers$ cd ..
arrakis:~$
```

(note el espacio entre "`cd`" y "`..`"). Cada directorio tiene una entrada de nombre "`..`" la cual se refiere al directorio padre. De igual forma, existe en cada directorio la entrada "`.`" la cual se refiere a sí mismo. Así que el comando

```
arrakis:~/papers$ cd .
arrakis:~/papers$
```

nos deja donde estamos.

También pueden usarse nombres con el camino absoluto en la orden `cd`. Para ir al directorio de Carlos con `cd`, introduciremos la siguiente orden.

```
arrakis:~/papers$ cd /home/carlos
arrakis:/home/carlos$
```

El uso de `cd` sin argumentos nos llevará a nuestro directorio de origen.

```
arrakis:/home/carlos$ cd
arrakis:~$
```

Otro truco es poner como argumento el signo "–"

```
arrakis:~$ cd --
arrakis:/home/carlos$
```

el cual nos lleva al último diretorio vitado (`/home/carlos` en este caso).

En los ejemplos que hemos visto, el prompt indicaba el camino absoluto al directorio en que nos encontrabamos, o lo refería a nuestro "home". Esto no ocurre

en todos los sistemas; pudiendo suceder que muestre sólo el último directorio, o no muestre nada.

Sea o no este el caso, tenemos un comando que nos muestra el directorio de trabajo actual: este es `pwd` (del ingles *Print name of current/Working Directory*) y puede resultarnos muy útil. Veamos cómo funciona:

```
arrakis:/home/carlos$ cd
arrakis:~$ pwd
/home/nahuel
arrakis:~$
```

1.3.2 Mirando el contenido de los directorios

Ahora que ya sabe cómo moverse por los directorios probablemente pensará: ¿Y bien? El simple movimiento por el árbol de directorios es poco útil; necesitamos un nuevo comando: `ls`. Éste muestra por la terminal la lista de archivos y directorios, por defecto, los del directorio activo. Por ejemplo:

```
arrakis:~$ ls
Mail     letters  papers
arrakis:~$
```

Aquí podemos ver que Nahuel tiene tres entradas en su directorio actual: `Mail`, `letters` y `papers`. Esto no nos dice demasiado; ¿son archivos o directorios? Podemos usar la opción `-F` de la orden `ls` para obtener más información. Por lo general los sistemas Linux tienen incluida esta opción.

```
arrakis:~$ ls --F
Mail/     letters/  papers/
arrakis:~$
```

Por el caracter "`/`" añadido a cada nombre sabemos que las tres entradas son subdirectorios.

La respuesta a la orden `ls -F` puede también añadir al final de un nombre el caracter"`*`"o "`@`"; el "`*`" indica que es un archivo **ejecutable**, y la "`@`" indica que se trata de un enlace (éstos los veremos en la Sección 1.10 de la página 29). Si `ls -F` no añade nada, entonces es un archivo normal, es decir no es ni un directorio ni un ejecutable.

Por lo general cada orden UNIX puede tomar una serie de opciones definidas en forma de argumentos. Éstos usualmente comienzan con el caracter "`-`", como vimos antes con `ls -F`. La opción `-F` le dice a `ls` que de más información sobre el tipo de archivos—en este caso añadiendo un `/` detrás de cada nombre de un directorio.

Si a `ls` le pasamos un nombre del directorio, mostrará el contenido de ese directorio.

```
arrakis:~$ ls --F papers
english-lit     history-final   masters-thesis  notes/
arrakis:~$
```

Para ver un listado más interesante, veamos el contenido de directorio del sistema `/etc`.

```
arrakis:~$ ls /etc

DIR_COLORS          ftpgroups           lynx.cfg            quota.conf*
HOSTNAME            ftphosts            mail/               rc.d/
Muttrc              ftpusers            mail.rc             redhat-release
TextConfig          gettydefs           mailcap             resolv.conf
X11/                gpm-root.conf       mailcap.vga         rmt@
XF86Config          group               man.config          rmtab
XF86Config.bak      group-              mgetty+sendfax/     rpc
adjtime             group.OLD           midi/               screenrc
aliases             host.conf           mime.types          securetty
aliases.db          hosts               minicom.users       security/
amd.conf            hosts.allow         motd                sendmail.cf
at.deny             hosts.deny          mtab                sendmail.cw
bashrc              hosts.lpd           mtools.conf         services
comanche.conf       httpd/              nmh/                services.rpmorig
conf.linuxconf      inetd.conf          nsswitch.conf       shadow
conf.modules        info-dir            ntp/                shadow-
cron.daily/         initrunlvl@         ntp.conf            shells
cron.hourly/        inittab             pam.conf            skel/
cron.monthly/       inputrc             pam.d/              smrsh/
cron.weekly/        ioctl.save          passwd              snmp/
crontab             issue               passwd-             sysconfig/
csh.cshrc           issue.net           passwd.OLD          syslog.conf
default/            ld.so.cache         pcmcia/             termcap
dosemu.conf         ld.so.conf          ppp/                uucp/
dosemu.users        lilo.conf           printcap            vga/
export              lilo.conf.rpmsave   printcap.bak        wine.conf
exports             localtime@          profile             wine.conf.rpmorig
fdprm               login.defs          profile.d/          wtmplock
fstab               logrotate.conf      protocols           yp.conf
ftpaccess           logrotate.d/        psdevtab            ypserv.conf
ftpconversions      ltrace.conf         pwdb.conf           ytalkrc

arrakis:~$
```

◇ **Nota:** Los usuarios de MS-DOS, notarán que los nombres de los archivos pueden contener más de 8 caracteres y contener puntos en cualquier posición. Incluso es posible que un archivo contenga más de un punto en su nombre (como es el caso de `ld.so.conf`).

Vayamos al directorio raíz con "`cd ..`" y desde allí al directorio `/usr/bin`.

```
arrakis:~$ cd ..
arrakis:/home$ cd ..
arrakis:/$ cd usr
arrakis:/usr$ cd bin
arrakis:/usr/bin$
```

También podemos movernos dentro de directorios en múltiples pasos, como en `cd /usr/bin`.

Trate de moverse por varios directorios usando `ls` y `cd`. En algunos casos podrá encontrarse con el desagradable mensaje de error "`Permission denied`". Esto simplemente es debido a cuestiones de seguridad del UNIX. Para poder moverse o listar un directorio debe tener permisos para hacerlo. Hablaremos más sobre ello en la Sección 1.9.

1.3.3 Creando directorios nuevos

Es el momento de aprender a crear directorios. Para ello se usa la orden `mkdir`. Pruebe lo siguiente:

```
arrakis:~$ mkdir foo
arrakis:~$ ls -F
Mail/     foo/      letters/  papers/
arrakis:~$ cd foo
arrakis:~/foo$ ls
arrakis:~/foo$
```

Acaba de crear un directorio nuevo y moverse a él. Como no hay ningún archivo en el directorio nuevo, no nos muestra nada. Ahora veamos cómo copiar archivos desde un lugar a otro.

1.3.4 Copia de archivos

La copia de archivos es efectuada por la orden `cp`:

```
arrakis:~/foo$ cp /etc/termcap  .
arrakis:~/foo$ cp /etc/shells  .
arrakis:~/foo$ ls --F
shells     termcap
arrakis:~/foo$ cp shells bells
arrakis:~/foo$ ls --F
bells     shells     termcap
arrakis:~/foo$
```

La orden `cp` copia los archivos listados en la línea de comandos al archivo o directorio pasado como último argumento. Nótese cómo se usa el "`.`" para referirnos al directorio actual.

1.3.5 Moviendo archivos

La orden `mv` mueve archivos en lugar de copiarlos. La sintaxis es muy sencilla.

```
arrakis:~/foo$ mv termcap sells
arrakis:~/foo$ ls -F
bells   sells   shells
arrakis:~/foo$
```

Nótese como `termcap` ya no existe y en su lugar está el archivo `sells`. Esta orden puede usarse para renombrar archivos, como acabamos de hacer, pero también para mover archivos a directorios diferentes.

◇ **Nota:** `mv` y `cp` sobreescribirán los archivos destino (si ya existen) sin consultar. Sea **cuidadoso** cuando mueva un archivo a otro directorio: puede haber ya un archivo con el mismo nombre que será sobreescrito.

◇ **Nota:** los comandos `cp`, `rm` y `mv` poseen la opción `-i` (de interactive en ingles), con la cual se nos preguntará antes de borrar o sebreescribir un archivo. Si no está seguro de lo que va a hacer, úsela.

1.3.6 Borrando archivos y directorios

Para borrar un archivo, use la orden `rm`. ("`rm`" viene de "remove").

```
arrakis:~/foo$ rm bells sells
arrakis:~/foo$ ls -F
shells
arrakis:~/foo$
```

Nos hemos quedado sólo con el archivo `shells`, pero no nos quejaremos. Nótese que `rm` por defecto no preguntará antes de borrar un archivo (a menos que se use con la opción `-i`), por lo tanto, sea cuidadoso. En UNIX un archivo borrado es un archivo borrado, **no hay forma de recuperarlo**.

Una orden relacionada con `rm` es `rmdir`. Esta orden borra un directorio, pero sólo si está vacío. Si el directorio contiene archivos o subdirectorios, `rmdir` se quejará.

1.3.7 Mirando los archivos

Las órdenes `more`, `cat` y `less` son usadas para ver el contenido de archivos. `more` y `less` muestran el archivo pantalla a pantalla, mientras que `cat` lo muestra entero de una vez.

Para ver el contenido del archivo `shells` podemos usar la orden

```
arrakis:~/foo$ more shells
```

Por si está interesado en el contenido de `shells`, es una lista de intérpretes de comandos (conocidos como shells) válidos disponibles en el sistema. En la mayoría de los sistemas incluye `/bin/sh`, `/bin/bash` y `/bin/csh`. Hablaremos sobre los diferentes intérpretes de comandos en la sección 1.6 (página 19).

Durante la ejecución de `more` pulse `Space` para avanzar a la página siguiente y `b` para volver a la página anterior. Hay otros comandos disponibles, los citados son sólo los más básicos. `q` finalizará la ejecución de `more`.

Salga de `more` y pruebe `cat /etc/termcap`. El texto probablemente pasará demasiado rápido como para poder leerlo. El nombre "`cat`" viene de "concatenate", que es para lo que realmente sirve el programa. La orden `cat` puede ser usada para concatenar el contenido de varios archivos y guardar el resultado en otro archivo. Esto se discutirá más adelante en la sección 2.3 (página 50).

1.3.8 Obteniendo ayuda en línea

Prácticamente cada sistema UNIX, incluido Linux, proporciona una utilidad conocida como "páginas de manual". Estas páginas contienen documentación en línea para todas las órdenes del sistema, recursos, archivos de configuración, etc.

La orden usada para acceder a las páginas de manual es `man`. Por ejemplo, si está interesado en conocer otras opciones de la orden `ls`, puede escribir

```
arrakis:~$ man ls
```

y le será mostrada la página de manual para `ls`.

Por lo general, en Linux, las páguinas de manual son mostradas con `less` y no con `more` como en otros UNIX. Para salir del uso de `less` debe utilizarse la tecla `q`.

Desafortunadamente la mayoría de las páginas de manual han sido escritas por gente que ya conocía lo que la orden o recurso hacía, por esto, las páginas de manual usualmente sólo contienen detalles técnicos de la orden sin ningún tipo de guía de uso. Pese a esto, dichas páginas son una gran fuente de información que permiten refrescar la memoria si olvidamos la sintaxis de un comando. Igualmente, le darán mucha información sobre órdenes que no trataremos en este texto.

Le sugiero que pruebe `man` con los comandos que ya hemos tratado y con los que vayamos introduciendo. Notará que alguno de los comandos no tiene página de manual. Esto puede ser debido a diferentes motivos. En primer lugar, las páginas no han sido escritas aún (el Proyecto de Documentación de Linux es también el responsable de las páginas de manual). En segundo lugar, la orden puede ser interna del intérprete de comandos, o un alias (como los tratados en la Sección 1.2.3), en cuyo caso no tendrán una página propia. Un ejemplo es la orden `cd` la cual es interna del intérprete de comandos. El propio intérprete de comandos es quien procesa `cd`—no hay un programa separado.

1.4 Resumen de órdenes básicas

Esta sección introduce algunas de las órdenes básicas más útiles de un sistema UNIX, incluidas las que ya hemos tratado en las secciones anteriores.

Nótese que las opciones usualmente comienzan con "`-`" y en la mayoría de los casos se pueden añadir múltiples opciones de una letra con un único "`-`". Por ejemplo, en lugar de usar `ls -l -F` es posible usar `ls -lF`.

En lugar de listar todas las opciones disponibles para cada uno de los comandos solo hablaremos de aquellas más útiles o importantes. De hecho, la mayoría de las órdenes tiene un gran número de opciones (muchas de las cuales nunca usará). Puede usar `man` para ver las páginas de manual de cada orden, la cual mostrará la lista completa de opciones disponibles.

Nótese también que la mayoría de las órdenes toman una lista de archivos o directorios como argumentos, denotados como "⟨*archivo1*⟩ ... ⟨*archivoN*⟩". Por ejemplo, la orden `cp` toma como argumentos la lista de archivos a copiar, seguidos del archivo o directorio destino. Cuando se copia más de un archivo, el destino debe de ser un directorio.

`cd` — Cambia el directorio de trabajo actual.
Sintaxis: `cd` ⟨*directorio*⟩
⟨*directorio*⟩ es el directorio al que cambiamos. ("`.`" se refiere al directorio actual, "`..`" al directorio padre.)
Ejemplo: `cd ../foo` pone `../foo` como directorio actual.

`ls` — Muestra información sobre los archivos o directorios indicados.
Sintaxis: `ls` ⟨*archivo1*⟩ ⟨*archivo2*⟩ `...`⟨*archivoN*⟩
donde ⟨*archivo1*⟩ a ⟨*archivoN*⟩ son los archivos o directorios a listar.
Opciones: Hay más opciones de las que podría suponer. Las más usadas comúnmente son: `-F` (usada para mostrar información sobre el tipo de archivo), `-l` (da un listado "largo" incluyendo tamaño, propietario, permisos, etc. Trataremos esto en detalle más adelante.), y `-a` (muestra los archivos que comienzan con "`.`" que por

defecto no son listados, motivo por el cual se los conoce como ocultos).
Ejemplo: `ls -lF /home/nahuel` mostrará el contenido del directorio `/home/nahuel`.

`pwd` Muestra el directorio de trabajo actual.
Sintaxis: `pwd`
no posee argumentos.

`cp` Copia archivo(s) en otro archivo o directorio.
Sintaxis: `cp` ⟨*archivo1*⟩ ⟨*archivo2*⟩ ...⟨*archivoN*⟩ ⟨*destino*⟩
donde ⟨*archivo1*⟩ a ⟨*archivoN*⟩ son los archivos a copiar, y ⟨*destino*⟩ es el archivo o directorio destino.
Ejemplo: `cp ../frog paula` copia el archivo `../frog` al archivo o directorio `paula`.

`mv` Mueve archivo(s) a otro archivo o directorio. Es equivalente a una copia seguida del borrado del original. Puede ser usado para renombrar archivos, como el comando MS-DOS `RENAME`.
Sintaxis: `mv` ⟨*archivo1*⟩ ⟨*archivo2*⟩ ...⟨*archivoN*⟩ ⟨*destino*⟩
donde ⟨*archivo1*⟩ a ⟨*archivoN*⟩ son los archivos a "mover" y ⟨*destino*⟩ es el archivo o directorio destino.
Ejemplo: `mv ../frog rana` mueve el archivo `../frog` al archivo o directorio `rana`.

`rm` Borra archivos. Considere que cuando los archivos son borrados en UNIX, son irrecuperables (a diferencia de MS-DOS, donde usualmente se puede recuperar un archivo borrado).
Sintaxis: `rm` ⟨*archivo1*⟩ ⟨*archivo2*⟩ ...⟨*archivoN*⟩
donde ⟨*archivo1*⟩ a ⟨*archivoN*⟩ son los nombres de los archivos a borrar.
Opciones: `-i` pedirá confirmación antes de borrar un archivo.
Ejemplo: `rm -i /home/nahuel/paula /home/nahuel/frog` borra los archivos `paula` y `frog` en `/home/nahuel`.

`mkdir` Crea directorios nuevos.
Sintaxis: `mkdir` ⟨*dir1*⟩ ⟨*dir2*⟩ ...⟨*dirN*⟩
donde ⟨*dir1*⟩ a ⟨*dirN*⟩ son los directorios a crear.
Ejemplo: `mkdir /home/nahuel/test` crea el directorio `test` colgando de `/home/nahuel`.

`rmdir` Esta orden borra directorios vacíos. Al usar `rmdir`, el directorio de trabajo actual no debe estar dentro del directorio a borrar.
Sintaxis: `rmdir` ⟨*dir1*⟩ ⟨*dir2*⟩ ...⟨*dirN*⟩
donde ⟨*dir1*⟩ a ⟨*dirN*⟩ son los directorios a borrar.
Ejemplo: `rmdir /home/nahuel/papers` borra el directorio `/home/nahuel/papers` si está vacío.

`man` Muestra la página de manual del comando o recurso (cualquier utilidad del sistema que no es un comando, como funciones de

biblioteca[6]) dado. Sintaxis: `man` ⟨*comando*⟩
donde ⟨*comando*⟩ es el nombre del comando o recurso sobre el que queremos obtener la ayuda.
Ejemplo: `man ls` muestra ayuda sobre la orden `ls`.

`more` Muesta el contenido de los archivos indicados, una pantalla cada vez.
Sintaxis: `more` ⟨*archivo1*⟩ ⟨*archivo2*⟩ ...⟨*archivoN*⟩
donde ⟨*archivo1*⟩ a ⟨*archivoN*⟩ son los archivos a mostrar.
Ejemplo: `more papers/history-final` muestra por la terminal el contenido del archivo `papers/history-final`.

`cat` Oficialmente usado para concatenar archivos, `cat` también es usado para mostrar el contenido completo de un archivo de una vez.
Sintaxis: `cat` ⟨*archivo1*⟩ ⟨*archivo2*⟩ ...⟨*archivoN*⟩
donde ⟨*archivo1*⟩ a ⟨*archivoN*⟩ son los archivos a mostrar.
Ejemplo: `cat letters/from-mdw` muestra por la terminal el contenido del archivo `letters/from-mdw`.

`echo` Simplemente envía a la terminal los argumentos pasados.
Sintaxis: `echo` ⟨*arg1*⟩ ⟨*arg2*⟩ ...⟨*argN*⟩
donde ⟨*arg1*⟩ a ⟨*argN*⟩ son los argumentos a mostrar.
Ejemplo: `echo ''Hola mundo''` muestra la cadena de caracteres "`Hola mundo`".

`grep` Muestra todas las líneas de un archivo dado que coinciden con un cierto patrón.
Sintaxis: `grep` ⟨*patrón*⟩ ⟨*archivo1*⟩ ⟨*archivo2*⟩ ...⟨*archivoN*⟩
donde ⟨*patrón*⟩ es una expresión regular y ⟨*archivo1*⟩ a ⟨*archivoN*⟩ son los archivos donde buscar. Ejemplo:
`grep arrakis /etc/hosts`
mostrará todas las líneas en el archivo `/etc/hosts` que contienen la cadena "`arrakis`".

1.5 Explorando el sistema de archivos

El **sistema de archivos** es la colección de archivos y la jerarquía de directorios de su sistema. Le prometimos acompañarlo/a por el sistema de archivos, y ha llegado el momento.

A esta altura del capítulo Ud. tiene el nivel y conocimientos para entender lo que muestra la Figura 1.1 (página 6): un árbol de directorios.

Primero cambie al directorio raíz (`cd /`) y ejecute `ls -F`. Probablemente verá estos directorios[7]: `bin`, `dev`, `etc`, `home`, `iraf`, `lib`, `mnt`, `proc`, `root`, `tmp`, `user`, `usr`, y `var`.

Echemos un vistazo a cada uno de estos directorios.

[6] Mal llamada librería en algunos textos.
[7] Puede ver otros o incluso no ver todos. No se preocupe. Cada versión de Linux, y cada computadora, difiere de otra en algunos aspectos.

`/bin`

`/bin` es la abreviatura de "binaries", o ejecutables. Es donde residen la mayoría de los programas esenciales del sistema. Use la orden "`ls -F /bin`" para listar los archivos. Podrá ver algunas órdenes que reconocerá, como `cp`, `ls` y `mv`. Estos son los programas para estas órdenes. Cuando usa la orden `cp` está ejecutando el programa `/bin/cp`.

Usando `ls -F` verá que la mayoría de los archivos de `/bin` tienen un asterisco ("`*`") añadido al final de sus nombres[8]. Esto indica que son archivos ejecutables, como describe la Sección 1.3.2. También verá algunos que terminan en ("`@`"), que nos indica que se trata de enlaces, los que trataremos en la Sección 1.10

`/dev`

El siguiente es `/dev`. Echémosle un vistazo de nuevo con `ls -F`.

Los "archivos" en `/dev` son conocidos como **controladores de dispositivos (device drivers)**: son usados para acceder a los dispositivos del sistema y recursos, como discos rígidos, modems, memoria, etc. Por ejemplo, de la misma forma que puede leer datos de un archivo, puede leerlos desde la entrada del ratón (mouse) leyendo `/dev/mouse`.

Los archivos que comienzan su nombre con `fd` son controladores de disqueteras (floppy driver). `fd0` es la primera disquetera, `fd1` la segunda. Ahora, alguien astuto se dará cuenta de que hay más controladores de dispositivos para disqueteras de los que hemos mencionado. Estos representan tipos específicos de discos. Por ejemplo, `fd1H1440` accederá a discos de 3.5"de alta densidad en la disquetera 1.

Aquí tenemos una lista de algunos de los controladores de dispositivos más usados. Nótese que, incluso aunque puede que no tenga alguno de los dispositivos listados, tendrá entradas en `dev` de cualquier forma.

- `/dev/console` hace referencia a la consola del sistema— es decir, al monitor conectado directamente a su sistema.
- Los dispositivos `/dev/ttyS` y `/dev/cua` son usados para acceder a los puertos serie. Por ejemplo, `/dev/ttyS0` hace referencia a "`COM1`" bajo MS-DOS. Los dispositivos `/dev/cua` son "callout", los cuales son usados simultaneámente con los `/dev/ttyS0` por dispositivos como un modem. En la próxima generación del kernel [9] de Linux (2.2.x) ambos están integrados, desapareciendo `/dev/cua`.
- Los nombres de dispositivo que comienzan por `hd` acceden a discos duros (del tipo IDE). `/dev/hda` hace referencia a la *totalidad* del primer disco duro, mientras que `/dev/hda1` hace referencia a la primera *partición* en `/dev/hda`.

[8] Recuerde que éste no forma parte del nombre.

[9] El *kernel* es el **núcleo** del sistema operativo; la parte del software que administra (o controla) todos los recursos físicos de la computadora.

- Los nombres de dispositivo que comienzan con `sd` son dispositivos SCSI. Si tiene un disco duro SCSI, en lugar de acceder a él mediante `/dev/hda`, deberá acceder a `/dev/sda`. Las cintas SCSI son accedidas vía dispositivos `st`, los CD-ROM SCSI vía `sr` y los Zip Drivers vía `sda4`, por lo general.
- Los nombres que comienzan por `lp` acceden a los puertos paralelo. `/dev/lp0` hace referencia a "`LPT1`" en el mundo MS-DOS.
- `/dev/null` es usado como "agujero negro"; cualquier dato enviado a este dispositivo desaparece. ¿Para qué puede ser útil esto?. Bien, si desea suprimir la salida por pantalla de una orden, podría enviar la salida a `/dev/null`. Hablaremos más sobre esto después (creo).
- Los nombres que comienzan por `/dev/tty` hacen referencia a "consolas virtuales" de su sistema (accesibles mediante las teclas `alt-F1`, `alt-F2`, etc). `/dev/tty1` hace referencia a la primera VC, `/dev/tty2` a la segunda, etc.
- Los nombres de dispositivo que comienzan con `/dev/pty` son "pseudo-terminales". Estos son usados para proporcionar una "terminal" a sesiones remotas. Por ejemplo, si su máquina está en una red, `telnet` de entrada usará uno de los dispositivos `/dev/pty`.

`/etc` — `/etc` contiene una serie de archivos de configuración del sistema. Estos incluyen `/etc/passwd` (la base de datos de usuarios), `/etc/rc.d` (script de inicialización del sistema), etc.

`/sbin` — `sbin` se usa para almacenar programas esenciales del sistema, que usará el administrador del sistema, y NO (por lo general) el usuario.

`/home` — `/home` contiene los directorios "home" de los usuarios. Por ejemplo, `/home/carlos` es el directorio del usuario "`carlos`". En un sistema recién instalado, no habrá ningún usuario en este directorio.

`/lib` — `/lib` contiene las **imágenes de las bibliotecas compartidas**. Estos archivos contienen código que compartirán muchos programas. En lugar de que cada programa contenga una copia propia de las rutinas compartidas, éstas son guardadas en un lugar común, en `/lib`. Esto hace que los programas ejecutables sean menores y reduce el espacio usado en disco.

`/proc` — `/proc` es un "sistema de archivos virtual". Los archivos que contiene realmente residen en memoria, no en un disco. Hacen referencia a varios **procesos** que corren en el sistema, y le permiten obtener información acerca de qué programas y procesos están corriendo en un momento dado. Entraremos en más detalles en la Sección 1.11.1.

`/tmp` — Muchos programas tienen la necesidad de generar cierta información temporal y guardarla en un archivo temporal. El lugar habitual para esos archivos es en `/tmp`.

`/usr` `/usr` es un directorio muy importante. Contiene una serie de subdirectorios que contienen a su vez algunos de los más importantes y útiles programas y archivos de configuración usados en el sistema.

Los directorios descritos arriba son esenciales para que el sistema esté operativo, pero la mayoría de las cosas que se encuentran en `/usr` son opcionales para el sistema. De cualquier forma, son estas cosas opcionales las que hacen que el sistema sea útil e interesante. Sin `/usr`, tendría un sistema aburrido, sólo con programas como `cp` y `ls`. `/usr` contiene la mayoría de los paquetes grandes de programas y sus archivos de configuración.

`/usr/X11R6` `/usr/X11R6` contiene el sistema X Window si usted lo instala. El sistema X Window es un entorno gráfico grande y potente el cual proporciona un gran número de utilidades y programas gráficos, mostrados en "ventanas" en su pantalla. Si está familiarizado con los entornos Microsoft Windows o Macintosh, X Window le será muy familiar. El directorio `/usr/X11R6` contiene todos los ejecutables de X Window, archivos de configuración y de soporte.

`/usr/bin` `/usr/bin` es el almacén real de programas del sistema UNIX. Contiene la mayoría de los programas que no se encuentran en otras partes como `/bin`.

`/usr/etc` Como `/etc` contiene diferentes archivos de configuración y programas del sistema, `/usr/etc` contiene incluso más que el anterior. En general, no son esenciales para el sistema, a diferencia de los que se encuentran en `/etc`, que sí lo son.

`/usr/include` `/usr/include` contiene los **archivos de cabecera** para el compilador de C. Estos archivos (la mayoría de los cuales terminan en `.h`, de "header") declaran estructuras de datos, subrutinas y constantes usadas en la escritura de programas en C. Los archivos que se encuentran en `/usr/include/sys` son generalmente usados en la programación en UNIX a nivel de sistema. Si está familiarizado con el lenguaje de programación C, aquí encontrará los archivos de cabecera como `stdio.h`, el cual declara funciones como `printf()`.

`/usr/g++-include`
`/usr/g++-include` contiene archivos de cabecera para el compilador de C++ (muy parecido a `/usr/include`).

`/usr/lib` `/usr/lib` contiene las librerías equivalentes "stub" y "static" a los archivos encontrados en `/lib`. Al compilar un programa, éste es "enlazado" con las librerías que se encuentran en `/usr/lib`, las cuales dirigen al programa a buscar en `/lib` cuando necesita el código de la librería. Además, varios programas guardan archivos de configuración en `/usr/lib`.

`/usr/local` `/usr/local` es muy parecido a `/usr`; contiene programas y archivos no esenciales para el sistema, pero que hacen el sistema más divertido y excitante. En general, los programas que se encuentran

en `/usr/local` son específicos de su sistema, esto es, el directorio `/usr/local` difiere bastante entre sistemas UNIX.

`/usr/man` Este directorio contiene las páginas de manual. Hay dos subdirectorios para cada "sección" de las páginas (use la orden `man man` para más detalles). Por ejemplo, `/usr/man/man1` contiene los fuentes (es decir, los originales por formatear) de las páginas de manual de la sección 1, y `/usr/man/cat1` las páginas ya formateadas de la sección 1. En versiones más modernas de Linux, las páginas formateadas pueden encontrarse en `/var/catman/`, pero esto no es algo que deba preocupar al usuario.

`/usr/src` `/usr/src` contiene el código fuente (programas por compilar) de varios programas de su sistema. El más importante es `/usr/src/linux`, el cual contiene el código fuente del Núcleo (Kernel) de Linux.

`/var` `/var` contiene directorios que a menudo cambian su tamaño o tienden a crecer. Muchos de estos directorios solían residir en `/usr`, pero desde que estamos tratando de dejarlo relativamente inalterable, los directorios que cambian a menudo han sido llevados a `/var`. Algunos de estos directorios son:

`/var/log` `/var/log` contiene varios archivos de interés para el administrador del sistema, específicamente históricos del sistema, los cuales recogen errores o problemas con el sistema. Otros archivos guardan las sesiones de presentación en el sistema, así como los intentos fallidos.

`/var/spool` `/var/spool` contiene archivos que van a ser pasados a otro programa. Por ejemplo, en una máquina conectada a una red, el correo de llegada será almacenado en `/var/spool/mail` hasta que el usuario destinatario lo lea o lo borre. Artículos nuevos de las "news" tanto salientes como entrantes pueden encontrarse en `/var/spool/news`; correo saliente que aún no pudo ser enviado puede encontrarse en `/var/spool/mqueue`, etc.

1.6 Tipos de intérpretes de comandos (shells)

Como hemos mencionado anteriormente en numerosas ocasiones, UNIX es un sistema operativo multitarea y multiusuario. La multitarea es *muy* útil, y una vez que la hayá probado, la usará continuamente. En poco tiempo podrá ejecutar programas "de fondo", conmutar entre múltiples tareas y "conectar" programas unos entre otros para conseguir resultados complejos con un único comando.

Muchas de las características que trataremos en esta sección son proporcionadas por el intérprete de comandos. Hay que tener cuidado en no confundir UNIX (el sistema operativo) con el intérprete de comandos; éste último, *es una interface con el sistema que hay debajo* (ésta es la definición de *shell*). El intérprete de comandos proporciona la funcionalidad sobre el UNIX.

El intérprete de comandos no es solo un intérprete interactivo de los comandos

que tecleamos, es también un potente lenguaje de programación, el cual permite escribir **script**[10], que a su vez permite juntar varias órdenes en un archivo. Los usuarios de MS-DOS reconocerán esto como los archivos "batch". El uso de los script del intérprete de comandos es una herramienta muy potente que le permitirá automatizar e incrementar el uso de UNIX (ver la Sección 1.13.1 para más información).

Hay varios tipos de intérpretes de comandos en el mundo UNIX. Los dos más importantes son el "Bourne shell" y el "C shell". El intérprete de comandos Bourne usa una sintaxis de comandos como la de los primeros sistemas UNIX, como el System III. El nombre del intérprete Bourne en la mayoría de los UNIX es `/bin/sh` (donde `sh` viene de "shell", cascara en castellano). El intérprete C usa una sintaxis diferente, a veces parecida a la del lenguaje de programación C, y en la mayoría de los sistemas UNIX se encuentra como `/bin/csh`.

Bajo Linux hay algunas diferencias en los intérpretes de comandos disponibles. Dos de los más usados son el "Bourne Again Shell" o "Bash" (`/bin/bash`) y Tcsh (`/bin/tcsh`). Bash es un equivalente al Bourne con muchas características avanzadas del C shell. Como Bash es un superconjunto de la sintaxis del Bourne, cualquier guión escrito para el intérprete de comandos Bourne estándar funcionará en Bash. Para los que prefieren el uso del intérprete de comandos C, Linux tiene el Tcsh, que es una versión extendida del C original y es la utilizada en los script del paquete de reducción de datos IRAF.

En casi todos los sistemas Linux se utiliza por defecto el intérprete de comandos bash. El tipo de intérprete de comandos que decida usar es puramente una cuestión de gustos. Algunas personas prefieren la sintaxis del Bourne con las características avanzadas que proporciona Bash, y otros prefieren el más estructurado intérprete de comandos C. En lo que respecta a los comandos usuales como `cp`, `ls`, etc, es indiferente el tipo de intérprete de comandos usado; la sintaxis es la misma. Sólo cuando se escriben script para el intérprete de comandos, o se usan características avanzadas, aparecen las diferencias entre los diferentes intérpretes de comandos.

Como estamos discutiendo sobre las diferencias entre los intérpretes de comandos Bourne y C, abajo veremos esas diferencias. Para los propósitos de esta guía, la mayoría de las diferencias son mínimas. (Si Ud. es realmente curioso/a a este respecto, lea las páginas de manual para `bash` y `tcsh`).

1.7 Caracteres comodín

Una característica importante de la mayoría de los intérpretes de comandos en UNIX es la capacidad para referirse a más de un archivo usando caracteres especiales. Estos llamados **comodines** le permiten referirse a, por ejemplo, todos los archivos que contienen el caracter "`n`".

El comodín "`*`" hace referencia a cualquier caracter o cadena de caracteres en el nombre del archivo. Por ejemplo, cuando usa el caracter "`*`" en el nombre de un archivo, el intérprete de comandos lo sustituye por todas las combinaciones posibles provenientes de los archivos en el directorio al cual nos estamos refiriendo.

Veamos un ejemplo rápido. Supongamos que Nahuel tiene los archivos `frog`, `joe` y `stuff` en el directorio actual.

```
arrakis:~$ ls
```

[10] En castellano sería **guiones**, pero en este texto se usará la palabra en ingles **script**

```
frog     joe     stuff
arrakis:~$
```

Para aceder a todos los archivos con la letra "o" en su nombre, hemos de usar la orden

```
arrakis:~$ ls *o*
frog     joe
arrakis:~$
```

Como puede ver, el comodín "`*`" ha sido sustituido con todas las combinaciones posibles que ajustan contra lo pedido (que contengan una "o") en el directorio actual.

El uso de "`*`" solo, simplemente se refiere a todos los archivos, puesto que todos los caracteres coinciden con el comodín.

```
arrakis:~$ ls *
frog     joe     stuff
arrakis:~$
```

Veamos unos pocos ejemplos más:

```
arrakis:~$ ls f*
frog
arrakis:~$ ls *ff
stuff
arrakis:~$ ls *f*
frog     stuff
arrakis:~$ ls s*f
stuff
arrakis:~$
```

El proceso de la sustitución de "`*`" en nombres de archivos es llamado **expansión de comodines** y es efectuado por el intérprete de comandos. Esto es importante: las órdenes individuales, como `ls`, *nunca* ven el "`*`" en su lista de parámetros. Es el intérprete quien expande los comodines para incluir todos los nombres de archivos que se adaptan. Luego la orden

```
arrakis:~$ ls *o*
```

es expandida para obtener

```
arrakis:~$ ls frog joe
```

Una nota importante acerca del caracter comodín "`*`".: El uso de este comodín *NO* cuadrará con nombres de archivos que comiencen con un punto ("`.`"). Estos archivos son tratados como "ocultos"; aunque no están realmente ocultos, simplemente no son mostrados en un listado normal de `ls` y no son afectados por el uso del comodín "`*`".

He aquí un ejemplo: Ya hemos mencionado que cada directorio tiene dos entradas especiales: "`.`", que hace referencia al directorio actual y "`..`", que se refiere al directorio padre. De cualquier forma, cuando use `ls` esas dos entradas no se mostrarán.

```
arrakis:~$ ls
frog     joe     stuff
arrakis:~$
```

Si usa el parámetro `-a` con `ls` podrá ver nombres de archivos que comienzan por "`.`". Observe:

```
arrakis:~$ ls -a
.     ..     .bash_profile     .bashrc     frog     joe     stuff
arrakis:~$
```

Ahora podemos ver las dos entradas especiales, "`.`" y "`..`", así como otros dos archivos "ocultos"—`.bash_profile` y `.bashrc`. Estos dos archivos son usados en el arranque por `bash` cuando Nahuel se presenta (loguea) al sistema; veremos más sobre éstos en la Sección 1.13.3.

Note que cuando usamos el comodín "`*`", no se muestra ninguno de los nombres de archivo que comienzan por "`.`".

```
arrakis:~$ ls *
frog     joe     stuff
arrakis:~$
```

Esto es una característica de seguridad: si "`*`" coincidiera con archivos que comienzan por "`.`" actuaría sobre "`.`" y "`..`". Esto puede ser peligroso con ciertas órdenes.

Otro caracter comodín es "`?`". Este caracter comodín sólo expande un único caracter. Luego "`ls ?`" mostrará todos los nombres de archivos con un carácter de longitud, y "`ls termca?`" mostrará "`termcap`" pero *no* "`termcap.backup`". Aquí tenemos otro ejemplo:

```
arrakis:~$ ls j?e
joe
arrakis:~$ ls f??g
frog
arrakis:~$ ls ????f
stuff
arrakis:~$
```

Como se puede ver, los caracteres comodín permiten referirse a más de un archivo a la vez. En el resumen de órdenes en la Sección 1.4 dijimos que `cp` y `mv` pueden copiar o mover múltiples archivos de una vez. Por ejemplo:

```
arrakis:~$ cp /etc/s* /home/nahuel
```

copiará todos los archivos de `/etc` que comiencen por "`s`" al directorio `/home/nahuel`. Por lo tanto, el formato de la orden `cp` es realmente

`cp` ⟨*archivo1*⟩ ⟨*archivo2*⟩ ⟨*archivo3*⟩ ...⟨*archivoN*⟩ ⟨*destino*⟩

donde ⟨*archivo1*⟩ a ⟨*archivoN*⟩ es la lista de los archivos a copiar, y ⟨*destino*⟩ es el archivo o directorio destino donde copiarlos. `mv` tiene idéntica sintaxis.

Nótese que si se está copiando o moviendo más de un archivo, ⟨*destino*⟩ debe ser un directorio. Sólo puede copiar o mover un *único* archivo a otro archivo.

◇ **Nota:** debe tenerse especial cuidado al utilizar comodines con el comando `rm`. Tenga en cuenta que la simple instrucción "`rm *`" *eliminará* en forma *definitiva* todos los archivos que se encuentren en el directorio actual (eceptuando los que comiencen con "."). No estará de más utilizar el comando `ls` con el juego de comodines escogido para asugurarse que la lista de archivos pedida corresponde a los que se desea eliminar.

1.8 Tubería UNIX

El título adecuado para esta sección, no es "Tubería UNIX"; los españoles dirían "Fontanería UNIX". Lo adecuado es "Pipes in UNIX", dejamos a Usted que lo interprete como le guste.

1.8.1 Entrada y salida estándar

Muchos comandos UNIX toman su entrada de algo conocido como **entrada estándar** y envían su salida a la **salida estándar** (a menudo abreviados como "stdin" y "stdout"). El intérprete de comandos configura el sistema de forma que la entrada estándar es el teclado y la salida la pantalla.

Veamos un ejemplo con el comando `cat`. Normalmente `cat` lee datos de los archivos cuyos nombres se pasan como argumentos en la línea de comandos y envía estos datos directamente a la salida estándar. Luego, usando el comando

```
arrakis:~/papers$ cat history-final masters-thesis
```

mostrará por pantalla el contenido del archivo `history-final` seguido por `masters-thesis`.

Si no se le pasan nombres de archivos a `cat` como parámetros, leerá datos de stdin y los enviará a stdout. Veamos un ejemplo.

```
arrakis:~/papers$ cat
Hola arrakis.  Enter
Hola arrakis.
Chau.  Enter
Chau.
ctrl-D
arrakis:~/papers$
```

Como se puede ver, cada línea que el usuario teclea (impresa en itálica) es inmediatamente reenviada al monitor por `cat`. Cuando se está leyendo de la entrada estándar, los comandos reconocen el fin de la entrada de datos cuando reciben el caracter EOT (end-of-text, fin de texto). Normalmente es generado con la combinación `ctrl-D`.

Veamos otro ejemplo. El comando `sort` toma como entrada líneas de texto (de nuevo leerá desde stdin si no se le proporcionan nombres de archivos en la línea de comandos), y devuelve la salida ordenada a stdout. Pruebe lo siguiente:

```
arrakis:~/papers$ sort
bananas
cocos
ajies
```

```
ctrl-D
ajies
bananas
cocos
arrakis:~/papers$
```

Podemos ordenar alfabéticamente la lista de la compra... ¿no es útil UNIX?

1.8.2 Redireccionando la entrada y salida

Ahora, supongamos que queremos que la salida de `sort` vaya a un archivo para poder salvar la lista ordenada de salida. El intérprete de comandos nos permite **redireccionar** la salida estándar a un archivo usando el símbolo "`>`". Veamos cómo funciona:

```
arrakis:~/papers$ sort > lista-compras
bananas
cocos
ajies
ctrl-D
arrakis:~/papers$
```

Como puede ver, el resultado de `sort` no se muestra por pantalla, en su lugar es salvado en el archivo `lista-compras`. Echemos un vistazo al archivo:

```
arrakis:~/papers$ cat lista-compras
ajies
bananas
cocos
arrakis:~/papers$
```

Ya podemos ordenar la lista de la compra y además guardarla. Imagino que a esta altura todos querrán tener una máquina Linux en su casa !!!

Supongamos ahora que teníamos guardada nuestra lista de compra desordenada original en el archivo `items`. Una forma de ordenar la información y salvarla en un archivo podría ser darle a `sort` el nombre del archivo a leer en lugar de la entrada estándar y redireccionar la salida estándar como hicimos arriba.

```
arrakis:~/papers$ sort items > lista-compras
arrakis:~/papers$ cat lista-compras
ajies
bananas
cocos
arrakis:~/papers$
```

Hay otra forma de hacer esto. No sólo puede ser redireccionada la salida estándar, también puede ser redireccionada la *entrada* estándar usando el símbolo "`<`".

```
arrakis:~/papers$ sort < items
ajies
bananas
cocos
arrakis:~/papers$
```

Técnicamente, `sort < items` es equivalente a `sort items`, pero nos permite demostrar que `sort < items` se comporta como si los datos del archivo fueran tecleados por la entrada estándar. El intérprete de comandos es quien maneja las redirecciones. `sort` no recibe el nombre del archivo (`items`) a leer, desde el punto de vista de `sort`, está leyendo datos de la entrada estándar como si fueran tecleados desde el teclado.

Esto introduce el concepto de **filtro**. Un filtro es un programa que lee datos de la entrada estándar, los procesa de alguna forma, y devuelve los datos procesados por la salida estándar. Usando la redirección, la entrada estándar y/o salida estándar pueden ser referenciadas desde archivos. `sort` es un filtro simple: ordena los datos de entrada y envía el resultado a la salida estándar. `cat` es incluso más simple: no hace nada con los datos de entrada; simplemente envía a la salida cualquier cosa que le llega.

1.8.3 Uso de tuberías (pipes)

Ya hemos visto cómo usar `sort` como un filtro. Pero estos ejemplos suponen que tenemos los datos en un archivo en alguna parte o vamos a introducir los datos manualmente por la entrada estándar. ¿Qué pasa si los datos que queremos ordenar provienen de la salida de otro comando, como `ls`?. Por ejemplo, usando la opción `-r` con `sort` ordenaremos los datos en orden inverso. Si queremos listar los archivos en el directorio actual en orden inverso, una forma podría ser la siguiente:

◇ **Nota:** seguramente `ls` tenga asignadas por defecto ciertas opciones como −−color y −F en la forma de un alias. Para no usar estas opciones puede usarse `\ls`, lo cual elimina los alias.

```
arrakis:~/papers$ ls
english-list
history-final
masters-thesis
notes
arrakis:~/papers$ \ls > file-list
arrakis:~/papers$ sort -r file-list
notes
masters-thesis
history-final
english-list
arrakis:~/papers$
```

Aquí salvamos la salida de `ls` en un archivo, y entonces ejecutamos `sort -r` sobre ese archivo. Pero esta forma necesita crear un archivo temporal en el que salvar los datos generados por `ls`.

La solución es usar las **pipes**[11]. El uso de pipes es otra característica del intérprete de comandos, que nos permite conectar una cadena de comandos en un "pipe", donde la stdout del primero es enviada directamente a la stdin del segundo y así sucesivamente. Queremos conectar la salida de `ls` con la entrada de `sort`. Para crear un pipe se usa el símbolo "`|`":

```
arrakis:~/papers$ ls | sort -r
notes
```

[11] N. del T.: tuberías

```
masters-thesis
history-final
english-list
arrakis:~/papers$
```

Esta forma es más corta y obviamente más fácil de escribir.

Otro ejemplo útil: usando el comando,

```
arrakis:~/papers$ ls /usr/bin
```

mostrará una larga lista de archivos, la mayoría de los cuales pasará rápidamente ante nuestros ojos sin que podamos leerlos. En lugar de esto, usemos `more` para mostrar la lista de archivos en `/usr/bin`.

```
arrakis:~/papers$ ls /usr/bin | more
```

Ahora podemos ir avanzando página a página cómodamente.

¡Pero la diversión no termina aquí!. Podemos "entubar" (o "conectar") más de dos comandos a la vez. El comando `head` es un filtro que muestra la primeras líneas del canal de entrada (aquí la entrada desde un pipe). Si queremos ver el último archivo del directorio actual en orden alfabético, usaremos:

```
arrakis:~/papers$ ls | sort -r | head -1
notes
arrakis:~/papers$
```

donde `head -1` simplemente muestra la primera línea de la entrada que recibe en este caso, el flujo de datos ordenados inversamente provenientes de `ls`).

1.8.4 Redirección no destructiva

El uso de ">" para redireccionar la salida a un archivo es destructivo: en otras palabras, el comando

```
arrakis:~/papers$ ls > file-list
```

sobreescribe el contenido del archivo `file-list`. Si en su lugar usamos el símbolo ">>", la salida será añadida al final del archivo nombrado, en lugar de ser sobreescrito.

```
arrakis:~/papers$ ls >> file-list
```

añadirá la salida de `ls` al final de `file-list`.

Es conveniente tener en cuenta que la redirección y el uso de pipes son características proporcionadas por el intérprete de comandos; éste proporciona estos servicios mediante el uso de la sintaxis ">", ">>" y "|".

1.9 Permisos de archivos

1.9.1 Conceptos de permisos de archivos

Al ser UNIX un sistema multiusuario, para proteger archivos de usuarios particulares de la manipulación por parte de otros, UNIX proporciona un mecanismo conocido como **permisos de archivos**. Este mecanismo permite que archivos y

directorios "pertenezcan" a un usuario en particular. Por ejemplo, como Nahuel creó archivos en su directorio "home", Nahuel es el propietario de esos archivos y tiene acceso a ellos.

UNIX también permite que los archivos sean compartidos entre usuarios y grupos de usuarios. Si Nahuel lo desea, podría restringir el acceso a sus archivos de forma que ningún otro usuario tenga acceso. De cualquier modo, por defecto, en la mayoría de los sistemas se permite que otros usuarios puedan leer tus archivos pero no modificarlos o borrarlos.

Como hemos explicado arriba, cada archivo pertenece a un usuario en particular. Por otra parte, los archivos también pertenecen a un **grupo** en particular, que es un conjunto de usuarios definido por el sistema. Cada usuario pertenece al menos a un grupo cuando es creado. El administrador del sistema puede hacer que un usuario tenga acceso a más de un grupo.

Los grupos usualmente son definidos por el tipo de usuarios que acceden a la computadora. Por ejemplo, en un sistema UNIX de una universidad, los usuarios pueden ser divididos en los grupos `estudiantes`, `dirección`, `profesores` e `invitados`. Hay también unos pocos grupos definidos por el sistema (como `bin` y `admin`) los cuales son usados por el propio sistema para controlar el acceso a los recursos; muy raramente los usuarios normales pertenecen a estos grupos.

Los permisos están divididos en tres tipos: lectura (read), escritura (write) y ejecución (execute). Estos permisos son dados para tres clases de usuarios: el propietario del archivo (user), el grupo al que pertenece el archivo (group) y para todos los usuarios independientemente del grupo, que podemos llamar el resto del universo (others).

El permiso de lectura permite a un usuario leer el contenido del archivo o, en el caso de un directorio, listar el contenido del mismo (usando `ls`). El permiso de escritura permite a un usuario escribir y modificar el archivo. Para directorios, el permiso de escritura permite crear nuevos archivos o borrar archivos ya existentes en dicho directorio. Por último, el permiso de ejecución permite a un usuario ejecutar el archivo si es un programa o script del intérprete de comandos. Para directorios, el permiso de ejecución permite al usuario cambiar al directorio en cuestión con `cd`.

◇ **Nota:** No se puede acceder a un directorio al que no se tiene permiso de ejecución.

1.9.2 Interpretando los permisos de archivos

Veamos un ejemplo del uso de permisos de archivos. Usando el comando `ls` con la opción `-l` se mostrará un listado "largo" de los archivos, el cual incluye los permisos de archivos:

```
arrakis:~/foo$ ls -l stuff

-rw-r--r--   1 nahuel    users         505 Mar 13 19:05 stuff

arrakis:~/foo$
```

El primer campo impreso en el listado representa los permisos de archivos. El tercer campo es el propietario del archivo (`nahuel`), y el cuarto es el grupo al cual pertenece el archivo (`users`). Obviamente, el último campo es el nombre del archivo (`stuff`), y los demás campos los trataremos más adelante.

Este archivo pertenece a `nahuel` y al grupo `users`. Echemos un vistazo a los permisos. La cadena `-rw-r--r--` nos informa, por orden, de los permisos para el propietario, el grupo del archivo y cualquier otro usuario.

El primer caracter de la cadena de permisos ("`-`") representa el tipo de archivo. El "`-`" significa que es un archivo regular. Las siguientes tres letras ("`rw-`") representan los permisos para el propietario del archivo, `nahuel`. El "`r`" para "lectura" y "`w`" para escritura. Luego, Nahuel tiene permisos de lectura y escritura para el archivo `stuff`.

Como ya mencionamos, aparte de los permisos de lectura y escritura está el permiso de "ejecución", representado por una "`x`". Como hay un "`-`" en lugar del "`x`", significa que Nahuel no tiene permiso para ejecutar ese archivo. Esto es correcto, puesto que `stuff` no es un programa de ningún tipo. Por supuesto, como el archivo es de Nahuel, puede darse a sí mismo permiso de ejecución, si lo desea. Esto será explicado en breve.

Los siguientes tres caracteres, `r--` representan los permisos para los miembros del grupo. El grupo al que pertenece el archivo es `users`. Como sólo aparece un "`r`" cualquier usuario que pertenezca al grupo `users` puede leer este archivo.

Los últimos tres caracteres, también `r--`, representan los permisos para cualquier otro usuario del sistema (diferentes del propietario o de los pertenecientes al grupo `users`). De nuevo, como sólo está presente el "`r`", los demás usuarios pueden leer el archivo, pero no escribir en él o ejecutarlo.

Aquí tenemos otros ejemplos de permisos de grupo:

`-rwxr-xr-x` El propietario del archivo puede leer, escribir y ejecutar el archivo. Los usuarios pertenecientes al grupo del archivo, y todos los demás usuarios pueden leer y ejecutar el archivo.

`-rw-------` El propietario del archivo puede leer y escribir. Nadie más puede acceder al archivo.

`-rwxrwxrwx` Todos los usuarios pueden leer, escribir y ejecutar el archivo.

1.9.3 Dependencias

Es importante darse cuenta de que los permisos de un archivo también dependen de los permisos del directorio en el que residen. Por ejemplo, aunque un archivo tenga los permisos `-rwxrwxrwx`, otros usuarios no podrán acceder a él a menos que también tengan permiso de lectura y ejecución para el directorio en el cual se encuentra el archivo. Si Nahuel quiere restringir el acceso a todos sus archivos, podría simplemente poner los permisos de su directorio "home" `/home/nahuel` a `-rwx------`. De esta forma ningún usuario podrá acceder a su directorio ni a ninguno de sus archivos o subdirectorios. Nahuel no necesita preocuparse de los permisos individuales de cada uno de sus archivos. Pero quien haga esto, puede decirse que tiene *cola de paja*, y no será el caso de Nahuel. El usuario que haga uso de correo electrónico, tendrá en su "home" el directorio `mail` (o `Mail`) con permisos `-rwx------`, lo que indica que ningún otro usuario podra ver su correo.

En otras palabras, para acceder a un archivo, hay que tener permiso de ejecución de todos los directorios a lo largo del camino de acceso al archivo, además de permiso de lectura (o ejecución) del archivo en particular.

Habitualmente, los usuarios de un sistema UNIX son muy abiertos con sus archivos. Los permisos que se dan a los archivos usualmente son `-rw-r--r--`, lo que

permite a todos los demás usuarios leer los archivos, pero no modificarlos de ninguna forma. Los directorios, usualmente tienen los permisos `-rwxr-xr-x`, lo que permite que los demás usuarios puedan moverse y ver los directorios, pero sin poder crear o borrar nuevos archivos en ellos.

Muchos usuarios pueden querer limitar el acceso de otros usuarios a sus archivos. Poniendo los permisos de un archivo a `-rw-------` no se permitirá a ningún otro usuario acceder al archivo. Igualmente, poniendo los permisos del directorio a `-rwx------` no se permitirá a los demás usuarios acceder al directorio en cuestión.

1.9.4 Cambiando permisos

El comando `chmod` se usa para establecer los permisos de un archivo. Sólo el propietario puede cambiar los permisos del archivo. La sintaxis de `chmod` es:

`chmod {a,u,g,o}{+,-}{r,w,x}` ⟨*filenames*⟩

Primero, indicamos a qué usuarios afecta **a**ll, **u**ser, **g**roup u **o**ther (primera llave). Después se especifica si se están añadiendo permisos (`+`) o quitándolos (`-`) (segunda llave). Finalmente se especifica qué tipo de permiso **r**ead, **w**rite o e**x**ecute (tercera llave) se modifica. Algunos ejemplos:

`chmod a+r stuff`
: Da a todos los usuarios acceso al archivo.

`chmod +r stuff`
: Como arriba, si no se indica `a`, `u`, `g` o `o` por defecto se toma `a`.

`chmod og-x stuff`
: Quita permisos de ejecución a todos los usuarios, excepto al propietario.

`chmod u+rwx stuff`
: Permite al propietario leer, escribir y ejecutar el archivo.

`chmod o-rwx stuff`
: Quita permisos de lectura, escritura y ejecución a todos los usuarios menos al propietario y a los usuarios del grupo del archivo.

1.10 Manejando enlaces (links) de archivos

Los links le permiten dar a un único archivo múltiples nombres. Los archivos son identificados por el sistema por su **número de inodo** [12], el cual es el único identificador del archivo para el sistema de archivos [13]. Un directorio es una lista de números de inodo con sus correspondientes nombres de archivo. Cada nombre de archivo en un directorio es un **link** a un inodo particular. Pero no se asuste, los **inodos** no son algo que deba preocuparnos por ahora.

[12] En ingles sería **inode**, lo cual es una palabra inventada que proviene de **I-node**, **un nodo tipo I** o **nodoi**; para nosotros sera **inodo**

[13] La orden `ls -i` mostrará los números de inodo.

1.10.1 Enlaces duros (Hard links)

La orden `ln` es usada para crear múltiples links para un archivo. Por ejemplo, supongamos que tiene un archivo `foo` en un directorio. Usando `ls -i`, veremos el número de inodo para el archivo.

```
arrakis:~$ ls -i foo
22192 foo
arrakis:~$
```

Aquí, el archivo `foo` tiene el número de inodo 22192 en el sistema de archivos. Podemos crear otro link a `foo`, llamado `bar`:

```
arrakis:~$ ln foo bar
```

Con `ls -i` veremos que los dos archivos tienen el mismo inodo.

```
arrakis:~$ ls -i foo bar
22192 bar   22192 foo
arrakis:~$
```

Ahora, accediendo a `foo` o a `bar` accederemos al mismo archivo. Si hace cambios en `foo`, estos cambios también serán efectuados en `bar`. Para todos los efectos, `foo` y `bar` son el mismo archivo.

Estos links son conocidos como *enlaces duros (hard links)* porque directamente crean el link al inodo. Nótese que sólo podemos crear hard links entre archivos del mismo sistema de archivos; los links simbólicos (ver más adelante) no tienen esta restricción.

Cuando borra un archivo con `rm`, está solamente borrando un link a un archivo. Si usa el comando

```
arrakis:~$ rm foo
```

sólo el link de nombre `foo` es borrado; `bar` todavía existirá. Un archivo es sólo definitivamente borrado del sistema cuando no quedan links a él. Usualmente, los archivos tienen un único link, por lo que el uso de `rm` los borra. Pero si el archivo tiene múltiples links, el uso de `rm` sólo borrará un único link; para borrar el archivo, deberá borrar todos los links del archivo.

La orden `ls -l` muestra el número de links a un archivo (entre otra información).

```
arrakis:~$ ls -l foo bar
-rw-r--r--   2 root     root          12 Aug  5 16:51 bar
-rw-r--r--   2 root     root          12 Aug  5 16:50 foo
arrakis:~$
```

La segunda columna en el listado, "2", especifica el número de links al archivo.

Así resulta que un directorio no es más que un archivo que contiene información sobre la translación link a inodo. También, cada directorio tiene al menos dos hard links en él: "`.`" (un link apuntando a sí mismo) y "`..`" (un link apuntando al directorio padre). En el directorio raíz (`/`), el link "`..`" simplemente apunta a `/`.

1.10.2 Enlaces simbólicos (Simbolic links)

Los enlaces simbólicos (simbolic links) son otro tipo de link, que es diferente al hard link. Un simbolic link permite dar a un archivo el nombre de otro, pero no enlaza el archivo con un inodo.

La orden `ln -s` crea un simbolic link a un archivo. Por ejemplo, si usamos la orden:

```
arrakis:~$ ln -s foo bar
```

crearemos un simbolic link `bar` apuntando al archivo `foo`. Si usamos `ls -i`, veremos que los dos archivos tienen inodos diferentes, en efecto:

```
arrakis:~$ ls -i foo bar
22195 bar   22192 foo
arrakis:~$
```

De cualquier modo, usando `ls -l` vemos que el archivo `bar` es un simbolic link apuntando a `foo`.

```
arrakis:~$ ls -l foo bar
lrwxrwxrwx   1 root     root           3 Aug  5 16:51 bar -> foo
-rw-r--r--   1 root     root          12 Aug  5 16:50 foo
arrakis:~$
```

Los bits de permisos en un simbolic link no se usan (siempre aparecen como `rwxrwxrwx`). En su lugar, los permisos del simbolic link son determinados por los permisos del archivo "apuntado" por el link (en nuestro ejemplo, el archivo `foo`).

Funcionalmente, los links duros y simbólicos son similares, pero hay algunas diferencias. Por una parte, se puede crear un simbolic link a un archivo que no existe; lo mismo no es cierto para hard links. Los simbolic links son procesados por el núcleo (kernel) de forma diferente a los duros (hard), lo cual es sólo una diferencia técnica, pero a veces importante. Los simbolic links son de ayuda puesto que identifican al archivo al que apuntan; con hard links no hay forma fácil de saber qué archivo está enlazado al mismo inodo.

Los links se usan en muchas partes del sistema Linux. Los links simbólicos son especialmente importantes para las imágenes de las bibliotecas compartidas en `/lib`.

1.11 Control de tareas

1.11.1 Tareas y procesos

El Control de Tareas es una utilidad incluída en muchas shells (incluídas **Bash** y **Tcsh**), que permite el control de multitud de comandos o **tareas** al momento. Antes de seguir, deberemos hablar un poco sobre los **procesos**.

Cada vez que usted ejecuta un programa, lanza lo que se conoce como *proceso*, que es simplemente el nombre que se le da a un programa cuando se está ejecutando. El comando `ps` visualiza la lista de procesos que se están ejecutando actualmente, por ejemplo:

```
arrakis:~$ ps
  PID TT STAT  TIME COMMAND
   24  3 S     0:03 (bash)
  161  3 R     0:00 ps
arrakis:~$
```

La columna `PID` representa el **identificador de proceso**. La última columna `COMMAND`, es el nombre del proceso que se está ejecutando. Ahora sólo estamos viendo los procesos que está ejecutando Nahuel[14]. Vemos que hay dos procesos, `bash` (que es el shell o intérprete de comandos que usa Nahuel), y el propio comando `ps`. Como puede observar, la `bash` se ejecuta concurrentemente con el comando `ps`. La `bash` ejecutó `ps` cuando Nahuel tecleó el comando. Cuando `ps` termina de ejecutarse (después de mostrar la tabla de procesos), el control retorna al proceso `bash`, que muestra el prompt, indicando que está listo para recibir otro comando.

Un proceso que está corriendo se denomina *tarea* para la shell. Los términos *proceso* y *tarea*, son intercambiables. Sin embargo, se suele denominar "tarea" a un proceso, cuando es usado en conjunción con **control de tareas**, que es un rasgo de la shell que permite cambiar entre distintas tareas.

En muchos casos, los usuarios sólo ejecutan un trabajo cada vez, que es el último comando que ellos teclearon desde la shell. Sin embargo, usando el control de tareas, usted podrá ejecutar diferentes tareas al mismo tiempo, cambiando entre cada una de ellas conforme lo necesite. ¿Cuán beneficioso puede llegar a ser esto? Supongamos que está usted con su procesador de textos, y de repente necesita parar y realizar otra tarea, con el control de tareas, usted podrá suspender temporalmente el editor, y volver a la shell para realizar cualquier otra tarea, y luego regresar al editor como si no lo hubiese dejado nunca. Lo siguiente sólo es un ejemplo, hay montones de usos prácticos del control de tareas.

1.11.2 Primer plano y segundo plano

Un proceso puede estar en **Primer plano** o en **Segundo plano**. Sólo puede haber un proceso en primer plano al mismo tiempo, el proceso que está en primer plano es el que interactúa con usted; recibe entradas de teclado, y envía las salidas al monitor (salvo, por supuesto, que haya redirigido la entrada o la salida, como se describe en la Sección 1.8). El proceso en segundo plano no recibe ninguna señal desde el teclado; por lo general, se ejecutan en silencio sin necesidad de interacción.

Algunos programas necesitan mucho tiempo para terminar, y no hacen nada interesante mientras tanto. Compilar programas es una de estas tareas, así como comprimir un archivo grande. No tiene sentido que se siente y se aburra mientras estos procesos terminan. En estos casos es mejor lanzarlos en segundo plano, para dejar a la computadora en condiciones de ejecutar otro programa.

Los procesos pueden ser **suspendidos**. Un proceso suspendido es aquel que no se está ejecutando actualmente, sino que está temporalmente parado. Después de suspender una tarea, puede indicar a la misma que continúe, en primer plano o en segundo, según necesite. Retomar una tarea suspendida no cambia en nada el estado de la misma—la tarea continuará ejecutándose justo donde se dejó.

Tenga en cuenta que suspender un trabajo no es lo mismo que *interrumpirlo*.

[14] Hay muchos más procesos aparte de éstos corriendo en el sistema. Para verlos todos, teclearemos el comando "`ps aux`".

Cuando usted interrumpe un proceso (generalmente con la pulsación de `ctrl-C`[15]), el proceso muere, y deja de estar en memoria y utilizar recursos de la computadora. Una vez eliminado, el proceso no puede continuar ejecutándose, y deberá ser lanzado otra vez para volver a realizar sus tareas. También se puede dar el caso de que algunos programas capturan la interrupción, de modo que pulsando `ctrl-C` no se detiene inmediatamente. Esto se hace para permitir al programa realizar operaciones necesarias de limpieza antes de terminar[16]. De hecho, algunos programas simplemente no se dejan matar por ninguna interrupción.

1.11.3 Envío a segundo plano y eliminación de procesos

Empecemos con un ejemplo sencillo. El comando `yes` es un comando aparentemente inútil que envía una serie interminable de `y`-es a la salida estándar. (Realmente es muy útil. Si se utiliza una tubería (o "pipe") para unir la salida de `yes` con otro comando que haga preguntas del tipo si/no, la serie de `y`-es confirmará todas las preguntas.)

Pruebe con esto.

```
arrakis:~$ yes
y
y
y
y
y
```

La serie de `y`-es continuará hasta el infinito, a no ser que usted la elimine, pulsando la tecla de interrupción, generalmente `ctrl-C`. También puede deshacerse de esta serie de `y`-es redirigiendo la salida estándar de `yes` hacia `/dev/null`, que como recordará es una especie de "agujero negro" o papelera para los datos. Todo lo que usted envíe allí, desaparecerá.

```
arrakis:~$ yes > /dev/null
```

Ahora va mucho mejor, la terminal no se ensucia, pero el prompt de la shell no retorna. Esto es porque `yes` sigue ejecutándose y enviando esas inútiles `y`-es a `/dev/null`. Para recuperarlo, pulse la tecla de interrupción.

Supongamos ahora que queremos dejar que el comando `yes` siga ejecutándose, y volver al mismo tiempo a la shell para trabajar en otras cosas. Para ello enviaremos a `yes` a segundo plano, lo que nos permitirá ejecutarlo, pero sin necesidad de interacción.

Una forma de mandar procesos a segundo plano es añadiendo un carácter "`&`" al final de cada comando.

```
arrakis:~$ yes > /dev/null &
[1] 164
arrakis:~$
```

Como podrá ver, ha regresado a la shell. ¿Pero, qué es eso de "`[1] 164`"?, ¿se está ejecutando realmente el comando `yes`?

[15] La tecla de interrupción puede definirse usando el comando `stty`. Por defecto, en la mayoría de sistemas es `ctrl-C`.

[16] Tiempo necesario para guardar algunos registros, etc.

"`[1]`" representa el **número de tarea** del proceso `yes`. La shell asigna un número a cada tarea que se esté ejecutando. Como `yes` es el único comando que se está ejecutando, se le asigna el número de tarea `1`. El número "`164`" es el número de identificación del proceso, o PID, que es el número que el sistema le asigna al proceso. Ambos números pueden usarse para referirse a la tarea como veremos después.

Ahora usted tiene el proceso `yes` corriendo en segundo plano, y enviando constantemente la señal `y` hacia el dispositivo `/dev/null`. Para chequear el estado del proceso, utilice el comando interno de la shell `jobs`:

```
arrakis:~$ jobs
[1]+  Running                 yes >/dev/null  &
arrakis:~$
```

¡Ahí está!. También puede usar el comando `ps`, como mostramos antes, para comprobar el estado de la tarea.

Para eliminar una tarea, utilice el comando `kill`. Este comando toma como argumento un número de tarea o un número de ID de un proceso. Esta era la tarea 1, así que usando el comando

```
arrakis:~$ kill %1
```

matará la tarea. Cuando se identifica la tarea con el número de tarea, se debe preceder el número con el caracter de porcentaje ("`%`").

Ahora que ya hemos matado la tarea, podemos usar el comando `jobs` de nuevo para comprobarlo:

```
arrakis:~$ jobs

[1]+  Terminated              yes >/dev/null

arrakis:~$
```

La tarea está, en efecto, muerta, y si usa el comando `jobs` de nuevo, no mostrará nada.

También podrá matar la tarea usando el número de ID de proceso (PID), el cual se muestra conjuntamente con el ID de tarea cuando arranca la misma. En nuestro ejemplo el ID de proceso es 164, así que el comando

```
arrakis:~$ kill 164
```

es equivalente a

```
arrakis:~$ kill %1
```

No es necesario usar el "`%`" cuando nos referimos a una tarea a través de su ID de proceso.

◇ **Nota:** si no especificamos otra cosa, `kill` mandará al proceso la señal `TERM` (equivalente a 15), la cual es recibida por el proceso, el que hace lo necesario para terminar su ejecución. El comando "`kill 164`" es equivalente a "`kill -15 164`". Una señal muy importante que podemos madar a un proceso es `KILL` (equivalente a 9); ésta, en general, no es atrapada por los procesos, sino manejada por el sistema y es éste el que termina con los procesos. Es especialmente útil, cuando un proceso ha dejado de funcionar correctamente o cuando queremos asegurarnos que el proceso terminará. En nuestro ejemplo pudimos haber usado el comando "`kill -9 164`" para matarlo.

1.11.4 Parada y relanzamiento de tareas

Hay otra manera de poner una tarea en segundo plano. Usted puede lanzarla como un proceso normal (en primer plano), pararla, y después relanzarla en segundo plano.

Primero, lance el proceso `yes` en primer plano como lo haría normalmente:

```
arrakis:~$ yes > /dev/null
```

De nuevo, dado que `yes` corre en primer plano, no debe retornar el prompt de la shell.

Ahora, en vez de interrumpir la tarea con `ctrl-C`, *suspenderemos* la tarea. El suspender una tarea no la mata: solamente la detiene temporalmente hasta que Ud. la retoma. Para hacer esto debe pulsar la tecla de suspender, que suele ser `ctrl-Z`.

```
arrakis:~$ yes > /dev/null
ctrl-Z
[1]+  Stopped                 yes >/dev/null
arrakis:~$
```

Mientras el proceso está suspendido, simplemente no se está ejecutando. No gasta tiempo de CPU en la tarea. Sin embargo, usted puede retomar el proceso de nuevo como si nada hubiera pasado. Continuará ejecutándose donde se dejó.

Para relanzar la tarea en primer plano, use el comando `fg` (del inglés "foreground").

```
arrakis:~$ fg
yes >/dev/null
```

La shell de nuevo muestra el nombre del comando, de forma que tenga conocimiento de qué tarea es la que ha puesto en primer plano. Pare la tarea de nuevo, con `ctrl-Z`. Esta vez utilice el comando `bg` para poner la tarea en segundo plano. Esto hará que el comando siga ejecutándose igual que si lo hubiese hecho desde el principio con "`&`" como en la sección anterior.

```
arrakis:~$ bg
[1]+ yes >/dev/null &
arrakis:~$
```

Y tenemos de nuevo el prompt. El comando `jobs` debería decirnos que `yes` se está ejecutando, y podemos matar la tarea con `kill` tal como lo hicimos antes.

¿Cómo podemos parar la tarea de nuevo? Si pulsa `ctrl-Z` no funcionará, ya que el proceso está en segundo plano. La respuesta es poner el proceso en primer plano de nuevo, con el comando `fg`, y entonces pararlo. Como puede observar podrá usar `fg` tanto con tareas detenidas, como con las que estén en segundo plano.

Hay una gran diferencia entre una tarea que se encuentra en segundo plano, y una que se encuentra detenida. Una tarea detenida es una tarea que no se está ejecutando, es decir, que no usa tiempo de CPU, y que no está haciendo ningún trabajo (la tarea aún ocupa un lugar en memoria, aunque puede ser volcada a disco). Una tarea en segundo plano, se está ejecutando, y usando memoria, a la vez que completando alguna acción mientras usted hace otro trabajo. Sin embargo, una tarea en segundo plano puede intentar mostrar texto en su terminal, lo que puede resultar molesto si está intentando hacer otra cosa. Por ejemplo, si usted usó el comando:

```
arrakis:~$ yes &
```

sin redirigir `stdout` a `/dev/null`, una cadena de `y`-es se mostrarán en su monitor, sin modo alguno de interrumpirlo (no puede hacer uso de `ctrl-C` para interrumpir tareas en segundo plano). Para poder parar esas interminables `y`-es, tendría que usar el comando `fg` para pasar la tarea a primer plano, y entonces usar `ctrl-C` para matarla.

Otra observación. Normalmente, los comandos `fg` y `bg` actúan sobre el último proceso parado (indicado por un "`+`" junto al número de tarea cuando usa el comando `jobs`). Si usted tiene varios procesos corriendo a la vez, podrá mandar a primer o segundo plano una tarea específica indicando el ID de tarea como argumento de `fg` o `bg`, como en

```
arrakis:~$ fg %2
```

(para la tarea de primer plano número 2), o

```
arrakis:~$ bg %3
```

(para la tarea de segundo plano número 3). No se pueden usar los ID de proceso con `fg` o `bg`.

Además de esto, el uso del número de tarea por sí sólo, como

```
arrakis:~$ %2
```

es equivalente a

```
arrakis:~$ fg %2
```

Sólo resta recordarle que el uso de control de tareas es una utilidad de la shell. Los comandos `fg`, `bg` y `jobs` son internos de la shell. Si por algún motivo usted utiliza una shell que no soporta control de tareas, no espere disponer de estos comandos.

Y además, hay algunos aspectos del control de tareas que difieren entre Bash y Tcsh. De hecho, algunas shells no proporcionan ningún control de tareas; sin embargo, la mayoría de las shells disponibles para Linux soportan control de tareas.

1.12 Usando el editor `vi`

Un **editor de texto** es simplemente un programa usado para la edición de archivos que contienen texto, como una carta, un programa en C, FORTRAN o un archivo de configuración del sistema. Mientras que hay muchos editores de texto disponibles en Linux, el único editor que está garantizado encontrar en cualquier sistema UNIX es `vi`; el "**vi**sual editor". `vi` no es el editor más fácil de usar, ni es muy autoexplicativo. De cualquier forma, como es tan común en el mundo UNIX y es posible que alguna vez necesite usarlo, aquí encontrará algo de documentación.

La elección de un editor es principalmente una cuestión de gusto personal y estilo. Muchos usuarios prefieren el barroco, autoexplicativo y potente **Emacs**; un editor con más características que cualquier otro programa, único en el mundo UNIX. Por ejemplo, Emacs tiene integrado su propio dialecto del lenguaje de programación LISP y tiene muchas extensiones . Pero como Emacs y todos sus archivos de soporte son relativamente grandes, puede que no tenga acceso a él en muchos sistemas.

vi, por otra parte, es pequeño y potente, pero más difícil de usar. De cualquier modo, una vez que conozca la forma de funcionamiento de vi, es muy fácil usarlo. Simplemente la curva de aprendizaje es bastante pronunciada al comienzo.

Esta sección es una introducción coherente a vi; no discutiremos todas sus características, sólo aquellas necesarias para que sepa cómo comenzar. Puede dirigirse a la página de manual de vi si está interesado en aprender más acerca de las características de este editor, o puede leer el libro *Learning the* vi *Editor* de O'Reilly and Associates.

1.12.1 Conceptos

Mientras se usa vi, en todo momento estará en uno de tres posibles modos de operación. Estos modos son conocidos como *modo órdenes*, *modo inserción* y *modo última línea*.

Cuando inicia vi, está en el *modo órdenes*. Este modo le permite usar ciertas órdenes para editar archivos o cambiar a otros modos. Por ejemplo, tecleando "x" mientras está en el modo órdenes, borra el caracter que hay debajo del cursor. Las teclas del cursor mueven a éste por el archivo que estamos editando. Generalmente, las órdenes usadas en este modo son sólo de uno o dos caracteres de longitud.

Habitualmente insertará o editará texto desde el *modo inserción*. Usando vi, probablemente dedicará la mayor parte del tiempo en este modo. Inicia el modo de inserción al usar una orden como "i" (para "insertar") desde el modo de órdenes. Una vez en el modo de inserción, irá insertando texto en el documento desde la posición actual del cursor. Para salir del modo de inserción y volver al de órdenes, pulse esc.

Modo última línea es un modo especial usado para proporcionar ciertas órdenes extendidas a vi. Al usar esos comandos, aparecen en la última línea de la pantalla (de ahí el nombre). Por ejemplo, cuando teclea ":" desde el modo de ódenes, entrará en el modo última línea, y podrá usar órdenes como "wq" (para escribir el archivo a disco y salir de vi), o "q!" (para salir de vi sin guardar los cambios). El modo de última línea es habitualmente usado por órdenes vi mayores de un carácter. En el modo de última línea, introduce una orden de una sola línea y pulsa enter para ejecutarla.

1.12.2 Comenzando con vi

La mejor forma de entender estos conceptos es arrancar vi y editar un archivo. En el ejemplo "screens" que veremos, vamos a mostrar (sólo) unas pocas líneas de texto, como si la pantalla tuviese sólo seis líneas de altura (en lugar de veinticuatro).

La sintaxis de vi es

```
vi ⟨archivo⟩
```

donde ⟨*archivo*⟩ es el nombre del archivo que desea editar.

Arranque vi tecleando

```
arrakis:~$ vi test
```

lo que editará el archivo test. Debería ver algo como

```
~
~
~
~
~
~
"test"[New file]
```

La columna de caracteres "~" indica que está al final del archivo.

1.12.3 Insertando texto

Está ahora en modo órdenes; para poder insertar texto en el archivo, pulse `i` (lo que le hará entrar en modo inserción), y comience a escribir.

```
Now is the time for all good men to come to the aid of the
party.
~
~
~
~
~
```

Mientras inserta texto, puede escribir tantas líneas como desee (pulsando `return` después de cada una, por supuesto), y puede corregir los errores con la tecla de borrado de caracter.

Para salir del modo de inserción y volver al modo de órdenes, pulse `esc`

Mientras esté en modo órdenes, puede usar las teclas del cursor para moverse por el archivo. En nuestro ejemplo, como solo tenemos una línea, el tratar de usar las teclas de línea arriba o abajo, probablemente hará que `vi` emita un pitido (bastante molesto).

Hay muchas formas de insertar texto aparte de la orden `i`. Por ejemplo, la orden `a` inserta texto comenzando *detrás* de la posición actual del cursor, en lugar de la posición actual del cursor. Para probar, use la tecla de cursor a la izquierda para desplazar el cursor entre las palabras "good" y "men".

```
Now is the time for all good men to come to the aid of the
party.
~
~
~
~
~
```

Pulse `a` para iniciar el modo inserción, teclee "`wo`" y pulse `esc` para volver al modo de ódenes.

```
Now is the time for all good women to come to the aid of the
party.
~
~
~
~
~
```

Para comenzar a insertar texto en la línea de abajo de la actual, use la orden "`o`". Por ejemplo, pulse `o` y teclee otra línea o más:

```
Now is the time for all good women to come to the aid of the
party.
Afterwards, we'll go out for pizza and beer.
~
~
~
~
```

Sólo recuerde que en cualquier momento está en modo de órdenes (donde órdenes como `i`, `a` o `o` son válidas), o en modo de inserción (cuando esté insertando texto, pulse `esc` para volver al modo de órdenes), o en modo de última línea (donde puede introducir comandos extendidos, como veremos más adelante).

1.12.4 Borrando texto

Desde el modo de órdenes, la orden `x` borra el caracter debajo del cursor. Si pulsa `x` cinco veces, terminará con:

```
Now is the time for all good women to come to the aid of the
party.
Afterwards, we'll go out for pizza and
~
~
~
~
```

Ahora pulse `a`, inserte algún texto, seguido de `esc`:

```
Now is the time for all good women to come to the aid of the
party.
Afterwards, we'll go out for pizza and Diet Coke.
~
~
~
~
```

Puede borrar líneas enteras usando la orden `dd` (es decir, pulse `d` dos veces en una fila). Si el cursor está en la segunda línea y teclea `dd`,

```
Now is the time for all good women to come to the aid of the
party.
~
~
~
~
~
```

Para borrar la palabra sobre la que se encuentra el cursor, use la orden `dw`. Sitúe el cursor sobre la palabra "good" y pulse `dw`.

```
Now is the time for all women to come to the aid of the
party.
~
~
~
~
~
```

1.12.5 Modificando texto

Puede sustituir secciones de texto usando la orden `R`. Sitúe el cursor en la primera letra de "party" y pulse [R], y escriba la palabra "hungry".

```
Now is the time for all women to come to the aid of the
hungry.
~
~
~
~
~
```

El uso de `R` para editar texto es bastante parecido al uso de las órdenes `i` y `a`, pero `R` sobreescribe texto en lugar de insertarlo.

La orden `r` sustituye un único caracter situado debajo del cursor. Por ejemplo, sitúe el cursor al comienzo de la palabra "Now" y escriba `r` seguido de `C`. Obtendrá:

```
Cow is the time for all women to come to the aid of the
hungry.
~
~
~
~
~
```

La orden "~" cambia de mayúsculas a minúsculas o viceversa la letra sobre la que se encuentra el cursor. Por ejemplo, si sitúa el cursor sobre la "o" de "Cow", y repetidamente pulsa [~], obtendrá:

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE
HUNGRY.
~
~
~
~
~
```

1.12.6 Órdenes de movimiento

Ya conoce cómo usar las teclas del cursor para moverse por el documento. Además, puede usar las órdenes `h`, `j`, `k` y `l` para mover el cursor a la izquierda, abajo, arriba y derecha respectivamente. Esto es muy cómodo cuando (por alguna razón) sus teclas de cursor no funcionen correctamente.

La orden `w` mueve el cursor al comienzo de la siguente palabra; `b` lo lleva al comienzo de la palabra anterior.

La orden `0` (cero) mueve el cursor al comienzo de la línea actual, y la orden `$` lo lleva al final de la línea.

Al editar archivos grandes, querrá moverse hacia adelante y atrás a lo largo del archivo mostrando una pantalla cada vez. Pulsando `ctrl-F` avanza el cursor una pantalla hacia adelante y `ctrl-B` lo lleva una pantalla atrás.

Para llevar el cursor al final del archivo, pulse `G`. También puede desplazarse a una línea arbitraria; por ejemplo, pulsando la orden `10G` llevará el cursor a la línea 10 del archivo. Para desplazarse al comienzo, use `1G`.

Puede asociar órdenes de desplazamiento con otras órdenes como es el borrado. Por ejemplo, la orden `d$` borrará todo desde la posición del cursor al final de la línea; `dG` borrará todo desde la posición del cursor al final del archivo.

1.12.7 Guardando archivos y saliendo de `vi`

Para salir de `vi` sin modificar el archivo use la orden `:q!`. Al pulsar "`:`" (*modo última línea*), el cursor se desplazará a la última línea de la pantalla; está en modo última línea.

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE
HUNGRY.
~
~
~
~
~
:_
```

En el modo de última línea hay disponibles una serie de órdenes extendidas. Una de ellas es `q!`, la cual permite salir de `vi` sin guardar los cambios. La orden `:wq` salva el archivo y sale de `vi`. La orden `ZZ` (desde el modo de órdenes, sin "`:`") es equivalente a `:wq`. Recuerde que debe pulsar `enter` después de introducir la orden para que ésta se ejecute en el modo última línea.

Para salvar el archivo sin salir de vi, simplemente use `:w`.

1.12.8 Editando otro archivo

Para editar otro archivo use la orden `:e`. Por ejemplo, para dejar de editar el archivo `test` y en su lugar editar el archivo `foo`, use la orden

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE
HUNGRY.
~
~
~
~
~
:e foo_
```

Si usa `:e` sin salvar primero el archivo, obtendrá el mensaje de error

```
No write since last change (":edit!" overrides)
```

lo cual significa que `vi` no quiere editar otro archivo hasta que salve el primero. En este punto, puede usar `:w` para guardar el archivo original, y entonces usar `:e`, o puede usar la orden

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE
HUNGRY.
~
~
~
~
~
:e!  foo_
```

El signo "`!`" le dice a `vi` lo que realmente desea usted; editar el nuevo archivo sin salvar los cambios del primero.

1.12.9 Incluyendo otros archivos

Si usa la orden `:r` puede incluir el contenido de otro archivo en el archivo que está editando. Por ejemplo

```
:r foo.txt
```

insertaría el contenido del archivo `foo.txt` en el texto en la posición actual de cursor.

1.12.10 Ejecutando comandos del intérprete

Puede también ejecutar comandos del intérprete desde el interior de `vi`. La orden `:r!` funciona como `:r`, pero en lugar de leer un archivo, inserta la salida de un comando dado en el archivo en la posición actual del cursor. Por ejemplo, si usa la orden

```
:r!  ls -F
```

obtendrá

```
COW IS THE TIME FOR ALL WOMEN TO COME TO THE AID OF THE
HUNGRY.
letters/
misc/
papers/
~
~
```

También puede salir a un intérprete de comandos desde `vi`, es decir, ejecutar una orden desde dentro de `vi` y volver al editor una vez que ésta finalice. Por ejemplo, si usa la orden

```
:!  ls -F
```

la orden `ls -F` será ejecutada, y los resultados mostrados en la pantalla, pero no insertados en el archivo en edición. Si usa la orden

```
:shell
```

`vi` iniciará una instancia del intérprete de comandos, permitiéndole temporalmente dejar a vi "parado" mientras ejecuta otras órdenes. Simplemente salga del intérprete de comandos (usando la orden `exit`) para regresar a `vi`.

1.12.11 Obteniendo ayuda

`vi` no proporciona demasiada ayuda de forma interactiva (la mayoría de los programas UNIX no lo hacen), pero siempre puede leer la página de manual para `vi`. `vi` es un "front-end" visual para el editor `ex`: es decir, es `ex` quien maneja la mayoría de las órdenes en el modo última línea. Luego además de leer la página de `vi`, consulte la de `ex` también.

1.13 Personalizando su entorno

El intérprete de comandos proporciona muchos mecanismos para personalizar su entorno de trabajo. Como hemos mencionado antes, el intérprete de comandos es más que un mero intérprete—es también un poderoso lenguaje de programación. Aunque escribir scripts del intérprete de comandos es una tarea extensa, nos gustaría describir algunas formas que pueden simplificar su trabajo en un sistema UNIX mediante el uso de características avanzadas del intérprete.

Como mencionamos antes, diferentes intérpretes usan diferentes sintaxis para la ejecución de scripts. Por ejemplo, *Tcsh* usa una notación al estilo C, mientras que Bourne usa otro tipo de sintaxis. En esta sección no nos fijaremos en las diferencias entre los dos y supondremos que los scripts son escritos con la sintaxis del intérprete de comandos Bourne.

1.13.1 Scripts del intérprete de comandos

Supongamos que usa una serie de comandos a menudo, y le gustaría acortar el tiempo requerido para teclear agrupándolos en una única "orden". Por ejemplo, las órdenes

```
arrakis:~$ cat chapter1 chapter2 chapter3 > book
arrakis:~$ wc -l book
arrakis:~$ lpr book
```

concatenará los archivos `chapter1`, `chapter2` y `chapter3` y guardará el resultado en el archivo `book`. Entonces, se mostrará el recuento del número de líneas del archivo `book` y finalmente se imprimirá con el comando `lpr`.

En lugar de teclear todos esos comandos, podría agruparlos en un **script del intérprete de comandos**. El script usado para ejecutar todas las órdenes sería

```
#!/bin/sh
# A shell script to create and print the book

cat chapter1 chapter2 chapter3 > book
wc -l book
lpr book
```

Si el script se salva en el archivo `makebook`, podría simplemente usar la orden

```
arrakis:~$ makebook
```

para ejecutar todas las órdenes del script. Los scripts son simples archivos de texto; puede crearlos con un editor como `emacs` o `vi` [17].

[17] `vi` se describió en la Sección 1.12.

Veamos este script. La primera línea "`#!/bin/sh`", identifica el archivo como un script y le dice al intérprete de comandos cómo ejecutarlo. Instruye al intérprete a pasarle el script a `/bin/sh` para la ejecución, donde `/bin/sh` es el programa del intérprete. ¿Por qué es esto importante? En la mayoría de los sistemas UNIX `/bin/sh` es un intérprete de comandos Bourne, como *Bash*. Forzando al script a ejecutarse usando `/bin/sh` nos estamos asegurando de que será interpretado según la sintaxis de Bourne. Esto hará que el script se ejecute usando la sintaxis Bourne aunque esté usando *Tcsh* como intérprete de comandos.

La segunda línea es un *comentario*. Estos comienzan con el caracter "`#`" y continúan hasta el final de la línea. Los comentarios son ignorados por el intérprete de comandos, son habitualmente usados para identificar el script con el programador.

El resto de las líneas del script son simplemente órdenes como las que podría teclear directamente. En efecto, el intérprete de comandos lee cada línea del script y ejecuta la línea como si hubiese sido tecleada en la línea de comandos.

Los permisos son importantes para los scripts. Si crea un script, debe asegurarse de que tiene permisos de ejecución para poder ejecutarlo[18]. La orden

```
arrakis:~$ chmod u+x makebook
```

puede ser usada para dar permisos de ejecución al script `makebook`.

1.13.2 Variables del intérprete de comandos y el entorno

El intérprete de comandos le permite definir **variables** como la mayoría de los lenguajes de programación. Una variable es simplemente un trozo de datos al que se le da un nombre.

◇ Nótese que *Tcsh*, así como otros intérpretes del estilo C, usan un mecanismo diferente al descrito aquí para inicializar variables. Esta discusión supondrá el uso del intérprete Bourne, como es *Bash* (el cual probablemente está usando). Vea la página de manual de *Tcsh* para más detalles.

Cuando asigna un valor a una variable (usando el operador "`=`"), puede acceder a la variable añadiendo a su nombre "`$`", como se ve a continuación:

```
arrakis:~$ foo=``hello there''
```

A la variable `foo` se le da el valor "`hello there`". Podemos ahora hacer referencia a ese valor a través del nombre de la variable con el prefijo "`$`". La orden

```
arrakis:~$ echo $foo
hello there
arrakis:~$
```

produce el mismo resultado que

```
arrakis:~$ echo ``hello there''
hello there
arrakis:~$
```

Estas variables son internas al intérprete. Esto significa que sólo éste podrá acceder a las variables. Esto puede ser útil en los scripts; si necesita mantener, por

[18] Cuando crea archivos de texto, los permisos por defecto usualmente no incluyen los de ejecución.

ejemplo, el nombre de un archivo, puede almacenarlo en una variable. Usando la orden `set` obtendrá una lista de todas las variables definidas en el intérprete de comandos.

De cualquier modo, el intérprete de comandos permite **exportar** variables al **entorno**. El entorno es el conjunto de variables a las cuales tienen acceso todas las órdenes que ejecute. Una vez que se define una variable en el intérprete, exportarla hace que se convierta también en parte del entorno. La orden `export` es usada para exportar variables al entorno.

◇ De nuevo, hemos de diferenciar entre *Bash* y *Tcsh*. Si está usando *Tcsh*, deberá usar una sintaxis diferente para las variables de entorno (se usa la orden `setenv`). Para más información, diríjase a la página de manual de *Tcsh*.

El entorno es muy importante en un sistema UNIX. Le permite configurar ciertas órdenes simplemente inicializando variables con las órdenes ya conocidas.

Veamos un ejemplo rápido. La variable de entorno `PAGER` es usada por la orden `man`. Especifica la orden que se usará para mostrar las páginas del manual una a una. Si inicializa `PAGER` con el nombre del programa, se usará éste para mostrar las páginas de manual en lugar de `less` (el cual es usado por defecto).

Inicialice `PAGER` a "`cat`". Esto hará que la salida de `man` sea mostrada de una vez, sin pausas entre páginas.

```
arrakis:~$ PAGER=''cat''
```

Ahora exportamos `PAGER` al entorno.

```
arrakis:~$ export PAGER
```

Pruebe la orden `man ls`. Las páginas deberían volar por su pantalla sin detenerse entre páginas.

Ahora, si inicializa `PAGER` a "`less`", se usará la orden `less` para mostrar las páginas del manual.

```
arrakis:~$ PAGER=''less''
```

Nótese que no vamos a usar la orden `export` después del cambio de la variable `PAGER`. Sólo hemos de exportar las variables una vez; cuaquier cambio efectuado con posterioridad será automáticamente propagado al entorno.

Las páginas de manual para una orden en particular, le informarán acerca del uso de alguna variable de entorno por parte de esa orden; por ejemplo, la página de manual de `man` explica que `PAGER` es usado para especificar la orden de paginado.

Algunas órdenes comparten variables de entorno; por ejemplo, muchas órdenes usan la variable `EDITOR` para especificar el editor por defecto para usar si es necesario.

El entorno es también usado para guardar información importante acerca de la sesión en curso. Un ejemplo es la variable de entorno `HOME`, que contiene el nombre del directorio de origen del usuario:

```
arrakis:~/papers$ echo $HOME
/home/nahuel
```

Otra variable de entorno interesante es `PS1`, la cual define el "prompt" principal que usará el intérprete. Por ejemplo:

```
arrakis:~$ PS1=''Your command, please:  ''
Your command, please:
```

Para volver a inicializar el "prompt" a su valor habitual (el cual contiene el directorio actual seguido por el símbolo "$"):

```
Your command, please:  PS1=''\h:\w\$ ''
arrakis:~$
```

La página de manual de `bash` describe la sintaxis usada para inicializar el "prompt".

1.13.2.1 La variable de entorno `PATH`

Cuando usa la orden `ls` ¿cómo encuentra el intérprete el programa ejecutable `ls`?. De hecho, `ls` se encuentra en `/bin/ls` en la mayoría de los sistemas. El intérprete usa la variable de entorno `PATH` para localizar los archivos ejecutables u órdenes que tecleamos.

Por ejemplo, su variable `PATH` puede inicializarse a:

```
/bin:/usr/bin:/usr/local/bin:.
```

Esto es una lista de directorios en los que el intérprere debe buscar. Cada directorio está separado por un "`:`". Cuando usa la orden `ls`, el intérprete primero busca `/bin/ls`, luego `/usr/bin/ls` y así hasta que lo localice o acabe la lista.

Nótese que `PATH` no interviene en la localización de archivos regulares. Por ejemplo, si usa la orden

```
arrakis:~$ cp foo bar
```

El intérprete no usará `PATH` para localizar los archivos `foo` y `bar`, esos nombres se suponen completos. Sólo se usará `PATH` para localizar el programa ejecutable `cp`.

Esto le permitirá ahorrar mucho tiempo; significa que no deberá recordar dónde son guardadas las órdenes. En muchos sistemas los archivos ejecutables se dispersan por muchos sitios, como `/usr/bin`, `/bin` o `/usr/local/bin`. En lugar de dar el nombre completo con el camino (como `/usr/bin/cp`), sólo hemos de inicializar `PATH` con la lista de los directorios donde queremos que se busquen automáticamente.

Nótese que `PATH` contiene "`.`", el cual es el directorio actual de trabajo. Esto le permite crear scripts o programas y ejecutarlos desde su directorio de trabajo actual sin tener que especificarlo directamente (como en `./makebook`). Si un directorio no está en su `PATH`, entonces el intérprete no buscará en él órdenes para ejecutar; ésto incluye al directorio de trabajo.

1.13.3 Scripts de inicialización del intérprete

Además de los scripts que puede crear, hay un número de éstos que usa el intérprete de comandos para ciertos propósitos. Los más importantes son sus **scripts de inicialización**, scripts automáticamente ejecutados por el intérprete al abrir una sesión.

Los scripts de inicialización son eso, simples scripts como los descritos arriba. Sin embargo, son muy útiles para la inicialización de su entorno al ejecutarse automáticamente.

Tanto *Bash* como *Tcsh* distinguen entre un **intérprete de presentación** y otras invocaciones del intérprete. Un intérprete de presentación es el que se ejecuta en el momento de la presentación al sistema (login). Es el único que usará. De cualquier modo, si ejecuta una opción de salir a un intérprete desde algún programa,

como `vi`, inicializa otra instancia del intérprete de comandos, el cual no es su intérprete de presentación. Además, en cualquier momento que ejecute un script, automáticamente estraá arrancando otro intérprete que va a ser el encargado de ejecutar el script.

Los archivos de inicialización usados por *Bash* son: `/etc/profile` (configurado por el administrador del sistema, y ejecutado por todos los usuarios de *Bash* en el momento de la presentación al sistema), `$HOME/.bash_profile` (ejecutado por una sesión de presentación *Bash*) y `$HOME/.bashrc` (ejecutado por todas las sesiones *Bash* que no son de presentación). Si `.bash_profile` no está presente, se usa en su lugar `.profile`

Tcsh usa los siguientes scripts de inicialización: `/etc/csh.login` (ejecutado por todos los usuarios de *Tcsh* en el momento de la presentación al sistema), `$HOME/.tcshrc` (ejecutado en la presentación al sistema por todas las instancias nuevas de *Tcsh*) y `$HOME/.login` (ejecutado en la presentación al sistema, seguido `.tcshrc`). Si `.tcshrc` no está presente, `.cshrc` se usa en su lugar.

Para entender completamente la función de estos archivos, necesitará aprender más acerca del intérprete de comandos. La programación de scripts es una materia complicada, más allá del alcance de esta guía. Lea las páginas de manual de `bash` y/o `tcsh` para aprender más sobre la configuración de su entorno.

1.14 ¿Quiere seguir por su cuenta?

Esperamos haberle proporcionado suficiente información para darle una idea básica de cómo usar el sistema, teniendo en cuenta que la mayoría de los aspectos más importantes e interesantes de Linux no están cubiertos aquí (esto es muy básico). Con esta base, en poco tiempo estará ejecutando complicadas aplicaciones y aprovechando todo el potencial del sistema. Si la cosa no es muy excitante al comienzo, no desespere: hay mucho que aprender.

Una herramienta indispensable para aprender acerca del sistema son las páginas del manual. Aunque muchas de las páginas pueden parecer confusas al principio, si se profundiza hay gran cantidad de información en ellas.

También es interesante leer un libro sobre la utilización de un sistema UNIX. Hay mucho más en UNIX de lo que pueda parecer a simple vista; desafortunadamente, la mayoría de ello queda fuera del alcance de esta guía.

Capítulo 2

Manejando Tablas en Linux

Autor: Federico Bareilles[1]

2.1 Introducción

En este capítulo veremos los comandos que podemos utilizar para el manejo de tablas en general, algunos de los cuales ya los hemos presentado en el Capítulo 1. Estos serán: `cat`, `head`, `tail`, `wc`, `grep`, `sort`, `cut`, `tr`, `awk` y `sed`. Después de leer esta lista podemos advertir que el UNIX es bastante parecido al lenguaje que hablaban los hombres de las cavernas (Figura[2] 2.1).

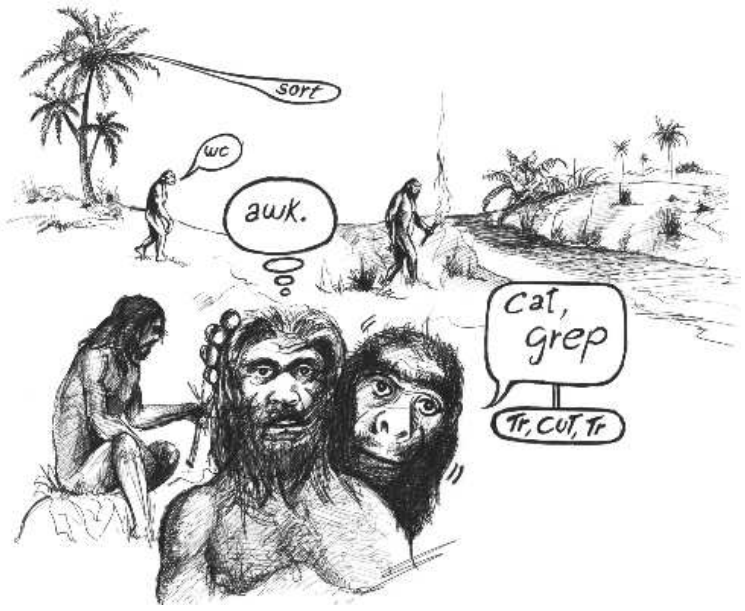


Figura 2.1: Orígenes del UNIX

[1] `fede@iar.unlp.edu.ar`
[2] Dibujo de Miguel Bareilles.

Consideraremos que una tabla es un conjunto de datos, ordenado en registros que serán las filas, y campos las columnas. En el resto del texto hablaremos de campos y registros en lugar de filas y columnas.

2.2 Ordenando registros con `sort`

Toma la entrada estándar (*stdin*) y la ordena en la salida estándar (*stdout*). Su nombre proviene de ordenar en ingles. Las opciones que pueden sernos de mayor utilidad son las siguientes:

- `-n`
  Comparar usando el orden numérico; especialmente util para ordenar números. Por defecto ordena alfabeticamente.
- `-r`
  Invertir el orden.
- `-u`
  Elimina lor renglones (regidtros) repetidos.
- `+POS1`
  Ordenar teniendo en cuenta para el orden lo que comienza en en el campo `POS1`).

2.3 `cat`

Su nombre proviene de concatenar (en inglés con**cat**enate, como vimos en 1.3.7), y lo que hace es concatenar archivos ("sumarlos") e imprimirlos por la salida estándar (*stdout*).

La primera impresión que nos da este comando es que no sirve para mucho, ya que lo que hace es conectar la entrada estándar (*stdin*) con la salida estándar (*stdout*); un simple tubo. La realidad es que es extremadamente útil.

En el siguiente ejemplo, utilizaremos como entrada estándar el teclado y, como salida, los archivos `lista-compra-1` y `lista-compra-2`:

```
arrakis:~$ cat > lista-compra-1
bananas
cocos
lechuga
ctrl-D
arrakis:~$ cat > lista-compra-2
zapallo
tomates
ajies
ctrl-D
arrakis:~$
```

donde redireccionamos la salida como se vio en la Sección 1.8.2 (página 24).

Usaremos `cat` para "sumar" los archivos `lista-compra-1` y `lista-compra-2`, direccionando la salida al comando `sort` para ordenar alfabéticamente el contenido de los dos archivos:

```
arrakis:~$ cat lista-compra-1 lista-compra-2 | sort
ajies
bananas
cocos
lechuga
tomates
zapallo
arrakis:~$
```

Podemos, también, guardar nuestra lista ordenada en un archivo redireccionando la salida de `sort` a éste:

```
arrakis:~$ cat lista-compra-1 lista-compra-2 | sort >
lista-compra
arrakis:~$
```

`cat` tiene algunas opciones que podrían sernos de gran utilidad; estas son:

- `-b, --number-nonblank`
  Numera todas las lineas que contengan texto (no las que tengan blancos).
- `-n, --number`
  Numera todas las lineas, incluidas las blancas. Ésto puede sernos muy util para imprimir código (FORTRAN por ejemplo) con el número de cada línea así: `cat -n archivo.f | lpr`.
- `-s, --squeeze-blank`
  Nunca mostrar más de una linea blanca en el caso de tener lineas blancas consecutivas.
- `-T, --show-tabs`
  Mostrar las tabulaciones como `^I`.
- `-E, --show-ends`
  Mostrar el fin de cada línea con el caracter $.

Podemos agregar el número de línea a nuestra lista de compras con `cat` así:

```
arrakis:~$ cat -n lista-compra
     1    ajies
     2    bananas
     3    cocos
     4    lechuga
     5    tomates
     6    zapallo
arrakis:~$
```

2.4 Contando palabras con wc

Este comando imprime el número de bytes, palabras y líneas de un archivo o de la entrada estándar (*stdin*). `wc` proviene del inglés **w**ord **c**ount[3].

Veamos cómo funciona:

[3] También suele llamarse **w**ater **c**loset.

```
arrakis:~$ wc lista-compra
      6       6      44 lista-compras
arrakis:~$
```

`wc` nos informa que `lista-compras` tiene 6 líneas, 6 palabras y 44 bytes de longitud. En este ejemplo, pudimos usar *cat lista-compras* | *wc*, con lo que obtendríamos el mismo resultado.

Las opciones de `wc` son:

- `-c, --bytes, --chars`
  Imprime sólo los bytes.

- `-w, --words`
  Imprime sólo la cantidad de palabras.

- `-l, --lines`
  Imprime sólo la cantidad de líneas.

2.5 Lenguaje de programación `awk`

Es el comando más complejo que trataremos en este capítulo, y es un lenguaje de programación en sí. Su nombre proviene de quienes lo crearon: **A**ho, **W**einberger y **K**ernighan[4].

Tenemos dos formas de programar para `awk`: una es escribir el programa en un archivo, y la otra es hacerlo directamente sobre la línea de comandos; esta última es la que usaremos en un comienzo.

2.5.1 Consideraciones generales sobre `awk`

En nuestro sistema Linux lo que en realidad usaremos es `gawk`: la versión *GNU* de `awk`. Por lo tanto podremos utilizar indistintamente `gawk` o `awk`, ya que este último es solamente un *simbolic link* a `gawk`, como se vio en la Sección 1.10.2 (página 31).

Tiene una sintaxis parecida a la del lenguaje **C**; así que si usted lo utiliza, `awk` le resultará familiar.

2.5.2 *"Hola mundo"* con `awk`

Como en cualquier lenguaje de programación, lo primero que haremos es hacer que imprima un simple cartelito que diga: "*Hola mundo*":

```
arrakis:~$ awk 'BEGIN{print "Hola mundo"}'
Hola mundo
```

simple ¿no? En la sección 2.5.4 tendremos una descripción más detallada de lo que pasa con el argumento de `awk`. Esperamos que este último comando resulte bastante autoexplicativo.

[4] Brian Kernighan junto con Dennis Ritchie participaron en la estandarización y creación del lenguaje de programación **C**.

2.5.3 Tipos de variables

En cuanto a los tipos de variables que utiliza, no es algo por lo que debamos preocuparnos, ya que se asignan automáticamente, e incluso los convierte según el entorno donde se esté utilizando la variable. Su inicialización tampoco debe preocuparnos, ya que es automática y depende del entorno. Veamos algunos ejemplos de cómo son interpretadas las variables según el entorno en el que son utilizadas:

- $a = a + 1$: (un simple contador) hará que a sea inicializada como numérica (tampoco importa si son enteros o punto flotante) con valor 0.
- $a = a$"$hola$": si aún nunca hemos utilizado a, se le asignará un string[5] vacío como ""; en cambio si a tenía un valor asignado como "nahuel", luego de la asignación tendrá el valor "nahuelhola".
- $a = 5$: a es numérica y contiene el valor 5.
  $a =$"$resultado$ "a: ¡epa! ¿ a no era numérica? sí, pero se la usa en otro contexto, por lo tanto se transforma en string y pasa a contener el valor "resultado 5". Lindo ¿ no ?

2.5.4 Modificar tablas con awk

Supongamos que tomamos el catálogo de estrellas brillantes *"Brigth Star Catalog"* (*BSC*)[6] (Hoffleit & Warren 1991, 5th Revised Ed., Astronomical Data Center, NSSDC/ADC), y de alguna forma hemos logrado contar cuántas estrellas hay de cada tipo espectral[7], sin importar la clase de luminosidad. Tendremos una tabla con los tipos espectrales: W, O, B, A, F, G, K, M, N, R, S, C y otros (-). Veamos nuestra tabla:

```
arrakis:~$ cat tipos.dat
 1 W    5
 2 O   51
 3 B 1758
 4 A 1966
 5 F 1307
 6 G 1233
 7 K 2236
 8 M  509
 9 N    1
10 R    0
11 S   10
12 C   19
13 -   15
```

Aquí tenemos una tabla de 3 campos con 13 registros: el primer campo es simplemente para numerar los tipos espectrales (sería útil si intentáramos graficar esto), el segundo contiene los tipos espectrales, y en el tercero tenemos la cantidad de estrellas[8] de ese tipo espectral que contiene el *BSC*.

[5] *string* es una cadena de caracteres; en el lenguaje FORTRAN se llaman *character*.

[6] Se puede obtener por ftp anónimo en: ftp://adc.gsfc.nasa.gov/pub/adc/archives/catalogs/5/5050/

[7] Si no tiene idea de lo que es un tipo espectral, no desespere, igual debería ser capaz de entender el ejemplo.

[8] Se ha contado solamente la componente primaria en el caso de sistemas múltiples.

Vamos ha realizar algo un poco más interesante con `awk`: contaremos el número total de estrellas en nuestra tabla. Simplemente sumaremos el campo **3** de todos los registros (sumamos la columna 3):

```
arrakis:~$ awk 'BEGIN{a = 0;} {a = a + $3;} END{print
a;}'  tipos.dat
9110
```

bien, tenemos 9110 estrellas en el catálogo.

Si se tiene alguna experiencia en programación, no debería amedrentarnos lo que acabamos de ver; sea o no éste el caso, desmenuzaremos el último comando:

- `'...'`: entre comillas simples introducimos el código que ejecutará `awk`
- `BEGIN{...}`: contiene el código que se ejecutará antes de procesar cualquier registro de entrada. En este caso es totalmente innecesario, ya que `awk` inicializa automáticamente las variables (se utilizó sólo por claridad).
- `{...}`: el par de llaves centrales contiene el código que se ejecuta por cada registro de entrada. Sólo usamos un contador que se incrementa con el número de estrallas.
- `END{...}`: el código que se ejecuta cuanado ya se ha procesado el último registro. Aquí mostramos el valor acumulado en a.
- **;**: cada orden es separada de otra por ";". En nuestro caso no era necesario utilizarlos porque tanto `BEGIN{}`, {}y `END{}` contienen una sola instrucción (u orden).
- **$3**: contiene el valor del campo **3**.

2.5.4.1 ¿Cómo funciona ésto?

Como hemos visto nuestro último comando pudo escribirse así:

```
arrakis:~$ awk '{a = a + $3} END{print a}'  tipos.dat
9110
```

`awk` lee el primer registro de `tipos.dat` (`1 W 5`), separa los campos y asigna:

Variable	Valor	Descripción
$0	␣1␣W␣␣␣␣5	contiene el registro completo (todos sus campos).
$1	1	contiene el campo número 1.
$2	W	contiene el campo número 2.
$3	5	contiene el campo número 3.
NF	3	contiene el número de campos leídos
NR	1	cuenta el número de registros

Con ␣ se denotan los espacios.

luego a se inicializa en 0, y se incrementa en 5. Continúa con los 12 registros siguientes incrementando a en $51, 1758, ..., 15$. Cuando ha terminado con el último registro, a contiene la suma total (9110). Al terminar con éstos, ejecuta lo que está dentro de `END{}` (que es simplemente imprimir el valor de a).

Falta mencionar que las variables **NF** y **NR**, junto con otras que aún no hemos introducido, son definidas internamente en `awk` y pueden sernos de gran utilidad.

◇ Un hecho implicito hasta este momento, del que no hemos hablado es ¿cómo distingue `awk` los diferentes campos? Existe una variable **FS** (field separator, separador de campos) para ello, que por defecto es cualquier número de espacios y de tabulaciones. De ser necesario, podremos cambiar este valor, para lo cual hay dos formas de hacerlo: una es poniendo $NF =$ "." dentro de `BEGIN{...}` (donde "." es el nuevo separador), y la otra es pasándolo como opción de `awk`: `awk -F "." '{...`

2.5.4.2 Dando formato a una tabla de LaTeX

Lo siguiente que haremos será modificar (con `awk`) la tabla de tipos espectrales para incluirla en un documento LaTeX. Conservaremos la segunda y tercera columna e incluiremos otra con los porcentajes de estrellas de cada tipo espectral. Para ello introduciremos las cosas de a una:

Comencemos por listar simplemente la segunda y tercer columna:

```
arrakis:~$ awk '{print $2, $3}'  tipos.dat
W 5
O 51
...
- 15
```

Notemos que al separar el argumento de *print* con comas se han intercalado espacios en la salida (`W␣5`, `O␣51`, ... y así siguiendo).

Lo siguiente será agregar el cálculo del porcentaje:

```
arrakis:~$ awk '{print $2, $3, $3 * 100 / 9110}'  tipos.dat
W 5 0.0548847
O 51 0.559824
...
- 15 0.164654
```

bien, tenemos los porcentajes pero sin formato; así no es muy agradable.

Agreguemos pues un formato para los valores en punto flotante. Recurriremos a la variable interna **OFMT** (*output format*); usaremos: $OFMT =$ "%2.2f"[9]. El valor para **OFMT** lo asignamos dentro de `BEGIN{}`:

```
arrakis:~$ awk 'BEGIN{OFMT="%2.2f"} {print $2, $3, $3 * 100 /
9110}'  tipos.dat
W 5 0.05
O 51 0.56
...
- 15 0.16
```

ahora está mejor.

Incluiremos el signo "&" para separar las columnas y la doble barra al final de cada registro[10]. Una forma simple de hacerlo sería simplemente incluirlo en la línea de *print* así: `print $2"&", $3"&", $3 * 100 / 9110,"\\\\"` (notar que se

[9] Los formatos tienen la misma sintaxis que la función *printf* del lenguaje **C**; puede encontrarse en su página de manual (3): `man 3 printf` o en la de `awk`: `man awk`.

[10] En LaTeX, "&" separa los campos y "\\" es el fin del registro.

debe introducir una doble barra para lograr una); pero hay una forma mucho más elegante que ésta y sin modificar la orden *print* que tenemos.

Utilizaremos la variable interna **OFS** (*output field separator*) que es la separación de campos para la salida (por defecto es un espacio, como hemos visto), y la variable interna **ORS** (*output record separator*) que es con lo que termina cada registro (Por defecto es el caracter de nueva línea "`\n`"):

```
arrakis:~$ awk 'BEGIN{OFMT="%2.2f"; OFS="&\t"; ORS="\\\\\n"}
{print $2, $3, $3 * 100 / 9110}'  tipos.dat
W&        5&      0.05\\
O&        51&     0.56\\
...
-&        15&     0.16\\
```

como separador de campos se usó también "`\t`", simplemente para incluir una tabulación; aunque no tenga ningún sentido para la tabla en LaTeX, sí lo tiene para un lector humano.

Utilizando ésto, y poniendo los comandos adecuados para definir la tabla, en el documento obtendremos el siguiente resultado:

```
\begin{tabular}{|c|r|r|}\hline
T.E.&     Cant.&  \%\\
\hline
W&        5&      0.05\\
O&        51&     0.56\\
B&        1758&   19.30\\
A&        1966&   21.58\\
F&        1307&   14.35\\
G&        1233&   13.53\\
K&        2236&   24.54\\
M&        509&    5.59\\
N&        1&      0.01\\
R&        0&      0\\
S&        10&     0.11\\
C&        19&     0.21\\
-&        15&     0.16\\
\hline
\end{tabular}
```

T.E.	Cant.	%
W	5	0.05
O	51	0.56
B	1758	19.30
A	1966	21.58
F	1307	14.35
G	1233	13.53
K	2236	24.54
M	509	5.59
N	1	0.01
R	0	0
S	10	0.11
C	19	0.21
-	15	0.16

Así se verá nuestra tabla

2.5.4.3 Una tabla un poco más complicada

Vamos a extraer del *BSC* la información necesaria para construir un diagrama HR [11].

El *BSC* tiene 9110 filas (como hemos visto), cada una de 197 caracteres de largo. El catálogo está armado para poder ser usado con rutinas en lenguaje FORTRAN; por lo tanto, para toda la información que en él hay, tenemos los formatos adecuados para su uso dados por sus autores en la documentación que lo acompaña. En la Tabla 2.1 listamos los primeros 10 registros del catálogo con el número de cada columna en forma vertical.

[11] Si no sabe de qué se trata, no se preocupe, intentaremos que pueda entender lo que se haga con `awk`.

```
                                                                                                     |  V  |
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111|11111|
000000000111111111122222222223333333333444444444455555555556666666666777777777788888888889999999999000|00000|
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012|34567|
------------------------------------------------------------------------------------------------------------
   1          BD+44 4550    3 36042          46          000001.1+444022000509.9+451345114.44-16.88| 6.70|
   2          BD-01 4525    6128569                      235956.2-010330000503.8-003011 98.33-61.14| 6.29|
   3 33    PscBD-06 6357   281285721002I        Var?     000013.0-061601000520.1-054227 93.75-65.93| 4.61|
   4 86    PegBD+12 5063   87 917012004                  000033.8+125023000542.0+132346106.19-47.98| 5.51|
   5          BD+57 2865  123 21085          61 V640 Cas 000101.8+575245000616.0+582612117.03-03.92| 5.96|
   6          CD-4914337  142214963      W               000108.4-493751000619.0-490430321.61-66.38| 5.70|
   7 10    CasBD+63 2107  144 109782005                  000114.4+633822000626.5+641146118.06  1.75| 5.59|
   8          BD+28 4704  166 73743          69    33    000125.2+282811000636.8+290117111.26-32.83| 6.13|
   9          CD-23    4  2031660531003                  000143.0-233947000650.1-230627 52.21-79.14| 6.18|
  10          BD-18 6428  256147090                      000211.8-175639000718.2-172311 74.36-75.90| 6.19|
  |(B-V)|                                        |  p  |
11|11111|11111111111111111111111111111111111111111111111|11111|1111111111111111111111111111111
01|11111|11111222222222233333333334444444444555555555566|66666|6667777777777888888888899999999
89|01234|56789012345678901234567890123456789012345678901|23456|7890123456789012345678901234567
-------------------------------------------------------------------------------------------------
  |+0.07| +0.08        A1Vn              -0.012-0.018 |      |-018      195  4.2  21.6AC   3
  |+1.10| +1.02       gG9                +0.045-0.060 |      |+014V
  |+1.04| +0.89 +0.54  K0IIIbCN-0.5      -0.009+0.089 |+.014|-006SB10 < 17  2.5   0.0     3*
  |+0.90|              G5III             +0.045-0.012 |      |-002V?
  |+0.67| +0.20        G5V               +0.263+0.030 |+.047|-012V           0.8   1.4      *
  |+0.52| +0.05        G1IV              +0.565-0.038 |+.050|+003SB          5.7   5.4      *
  |-0.03| -0.19        B9III            e+0.008 0.000 |      |-000V     153                  *
  |+0.75| +0.33        K0V               +0.380-0.182 |+.067|-008V           2.6 158.6AB   4*
  |+0.38| +0.05        A7V               +0.100-0.045 |      |+003V
  |+0.14| +0.10        A6Vn              -0.018+0.036 |      |-009V?    195                  *
```

Tabla 2.1: 10 registros del *BSC*

¿Qué elementos necesitamos para construir nuestros diagramas? Graficaremos el índice de color $(B-V)$ contra la magnitud visual absoluta M_V. $(B-V)$ está en el catálogo, pero M_V no, tendremos que calcularla. Lo haremos usando la paralaje y magnitud visual, de la siguiente manera:

$$M_V = V - 5 * log_{10}\left(\frac{1}{10 * p}\right) \tag{2.1}$$

donde V es la magnitud visual y p la paralaje (ambas son datos del catálogo). Entonces: extraigamos del catálogo la magnitud visual V, el índice de color $(B-V)$ y la paralaje p usando `awk`. De la descripción del catálogo encontramos que:

- V está entre la columna 103 y 107.
- $(B-V)$ entre 110 y 114.
- p entre 162 y 166.

En la Tabla 2.1, podemos ver que no todos los datos están presentes para todas las estrellas, por ejemplo la paralaje está presente para la 3, 5, 6 y 8 y no para 1, 2, 4, 7, 9 y 10. Incluso otros datos no están separados por ningún caracter, ya que está pensado para ser utilizado con formatos apropiados en lenguaje FORTRAN, pero nosotros (testarudos) lo haremos con `awk` de la siguiente manera:

Utilizaremos como separador de campos el caracter nulo "", con lo cual `awk` entenderá que cada columna es un campo, es decir, cada caracter pertenece a un campo, tendremos un total de 197 campos (correspondientes a los 197 caracteres de cada fila).

Como prueba extraigamos la magnitud visual V para ir entendiendo la idea. Ésta está entre las columnas 103 y 107, por lo que deberemos extraer todos los caracteres entre 103 y 107 para obtenerla, intentémoslo:

```
arrakis:~$ awk -F "" '{v=($103$104$105$106$107); print v }'
catalog.dat
 6.70
 6.29
 4.61
 ...
 6.25
 5.80
```

Extraer las magnitudes no parece complicado. Si entendemos cómo funciona esto, ahora hagámoslo con los 3 valores que necesitamos: V, $(B-V)$ y p. Esta vez no dejaremos que los datos fluyan por la consola (*stdout*), sino que, los dirigiremos a un archivo que llamaremos `v-bv-p.dat`:

```
arrakis:~$ awk -F "" '{v=($103$104$105$106$107);
bv=($110$111$112$113$114); p=($162$163$164$165$166);print
v,bv,p;}' catalog.dat > v-bv-p.dat
```

Ahora podemos echar un vistazo a nuestro nuevo archivo para ver qué forma tiene. Hágalo con `less` o con `more`.

Como suponíamos, nuestro nuevo archivo **no** posee los 3 valores extraídos para todas las estrellas. Esto lo deberemos tener en cuenta en el próximo paso cuando apliquemos la fórmula 2.1. Por suerte `awk` tiene una variable, ya vista, que nos será muy útil: **NF** (*number of fields*), que contiene el número de campos para cada registro que se lee, entonces pondremos como condición que **NF** sea igual a 3. Otro problema que surge de estudiar los valores que obtuvimos en `v-bv-p.dat` son paralajes negativas. Pues bien, eso no es correcto para una paralaje y se trata de estrellas cuya paralaje no pudo determinarse; deberemos entonces pedir que ésta sea positiva ($p > 0$) y descartar el resto.

¿Cómo imponemos condiciones? Muy simple, `awk` posee una instrucción *if* muy similar al lenguaje **C**, y se puede considerar, también, similar al bloque *if* del lenguaje FORTRAN:

```
if( condición ){ sentencias } else{ sentencias }
if( condición ){ sentencias }
if( condición ) sentencia;
```

En la Tabla 2.2 podemos ver algunos de los operadores disponibles en `awk`.

Usemos ésto para pedir ahora que cada registro contenga 3 campos y que la paralaje sea positiva, e incluyamos ya la cuenta de la fórmula 2.1:

```
arrakis:~$ awk '{if(NF == 3 && $3 >0 )print
$2,$1-5*log(1/(10*$3))/log(10)}' v-bv-p.dat > hr.dat
```

◇ *log*() en `awk`, como en FORTRAN, es el logaritmo natural y no $log_{10}()$ como uno podría suponer. Por este motivo dividimos por el $log(10)$.

Si todo anduvo bien, debemos tener en el archivo `hr.dat` el índice de color $(B-V)$ y la magnitud absoluta M_V; dé una mirada al archivo para ver si todo anduvo correctamente. Éste debe tener la siguiente forma:

Símbolo	Descripción
<	Menor
>	Mayor
<=	Menor igual
>=	Mayor igual
!=	Distinto
==	Igual
&&	And lógico
||	Or lógico

Tabla 2.2: Operadores en `awk`

```
+1.04 0.34064
+0.67 4.32049
+0.52 4.19485
+0.75 5.26037
-0.11 -0.41425
...
+1.63 0.9606
+1.07 1.23972
+0.62 3.84034
```

Grafiquemos esta tabla de números para ver qué forma tiene. Para hacerlo vamos a usar `gnuplot`[12], pero su uso no será cubierto en este capítulo. Si le interesa obtener documentación sobre éste, puede encontrarla en:

`ftp://picard.tamu.edu/pub/gnuplot`

Con algún editor de textos, creamos un archivo (que podemos llamar `hr.gnu`) que contenga el siguiente texto:

```
set terminal postscript
set output "hr.ps"
set size 1,1
set title "Diagrama HR"
set ylabel "Mv"
set xlabel "(B-V)"
set yrange [10:-10]
set nokey
plot "hr.dat" with points pt 7 ps 0.3
```

Luego generamos el gráfico con `gnuplot` de la siguiente manera: [13]

```
arrakis:~$ gnuplot hr.gnu
```

Obtendremos un archivo PostScript (`hr.ps`) como el de la Figura 2.2.

[12] Se pronuncia ñuplot.
[13] Otra forma es tipear cada comando dentro del interprete del `gnuplot`.

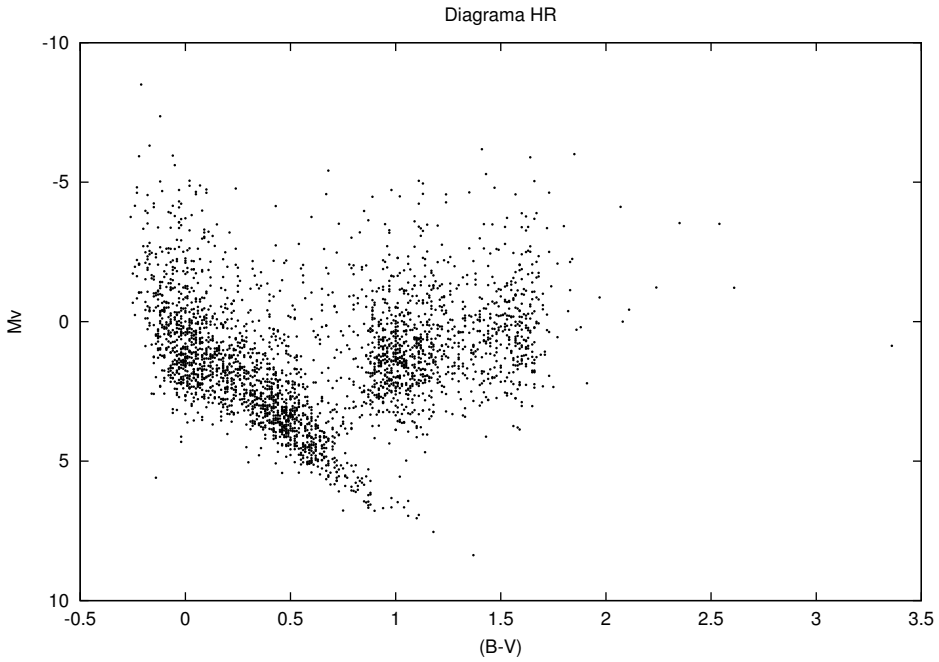


Figura 2.2: Diagrama HR

Apéndice A

Breve historia de Linux

UNIX es uno de los sistemas operativos más populares del mundo debido a su extenso soporte y distribución. Originalmente fue desarrollado como sistema multitarea con tiempo compartido para minicomputadoras y *mainframes* a mediados de los 70, y desde entonces se ha convertido en uno de los sistemas más utilizados a pesar de su, ocasionalmente, confusa interfaz con el usuario y el problema de su estandarización.

¿Cuál es la verdadera razón de la popularidad de UNIX? Muchos hackers[1] consideran que UNIX es el auténtico y único sistema operativo. El desarrollo de Linux parte de un grupo en expansión de hackers de UNIX que quisieron hacer su sistema con sus propias manos.

Existen numerosas versiones de UNIX para muchos sistemas, desde computadoras personales hasta supercomputadores como la Cray Y-MP. La mayoría de las versiones de UNIX para computadoras personales son muy caras. Cuando se escribía esto, una copia para una máquina 386 del UNIX System V de AT&T costaba unos 1500 dólares estadounidenses.

Linux es una versión de UNIX de libre distribución, inicialmente desarrollada por Linus Torvalds[2] en la Universidad de Helsinki, en Finlandia. Fue desarrollado con la ayuda de muchos programadores y expertos de UNIX a lo largo y ancho del mundo, gracias a la presencia de Internet. Cualquier habitante del planeta puede acceder a Linux y desarrollar nuevos módulos o cambiarlo a su antojo. El núcleo (*kernel*) de Linux no utiliza ni una sola línea del código de AT&T o de cualquier otra fuente de propiedad comercial, y buena parte del software para Linux se desarrolla bajo las reglas del proyecto de GNU de la Free Software Foundation, Cambridge, Massachusetts.

[1] **hacker**: [Originalmente, alguien que fabrica muebles con un hacha]. 1) Persona que disfruta con la exploración de los detalles de los sistemas programables y cómo aprovechar sus posibilidades, al contrario de la mayoría de los usuarios, que prefieren aprender sólo lo imprescindible. 2) El que programa de forma entusiasta (incluso obsesiva). 3) Persona que es buena programando de forma rápida. 4) Experto en un programa en particular, o que frecuentemente realiza trabajo usando cierto programa; como en "es un hacker de UNIX". (Las definiciones 1 a 4 están correlacionadas, y la gente que encaja en ellas suele congregarse). 5) Experto o entusiasta de cualquier tipo . Se puede ser un hacker astrónomo, por ejemplo. 6) El que disfruta del reto intelectual de superar o rodear las limitaciones de forma creativa. 7) [en desuso] Liante malicioso que intenta descubrir información sensible urgando por ahí. De ahí vienen "hacker de contraseñas" y "hacker de redes". El término correcto en estos casos es cracker.
Fuente: *"The New Hackers Dictionary . Eric S . Raymond."*

[2] `torvalds@kruuna.helsinki.fi`.

Inicialmente, sólo fue un proyecto de aficionado para Linus Torvalds. Se inspiraba en Minix, un pequeño UNIX desarrollado por Andy Tanenbaum, y las primeras discusiones sobre Linux surgieron en el grupo de News `comp.os.minix`. Estas discusiones giraban en torno al desarrollo de un pequeño sistema UNIX de carácter académico dirigido a aquellos usuarios de Minix que querían algo más.

El desarrollo inicial de Linux ya aprovechaba las características de conmutación de tareas en modo protegido del 386, y se escribió todo en código *assembler*. Linus dice:

> "Comencé a utilizar el C tras escribir algunos drivers, y ciertamente se aceleró el desarrollo. En este punto sentí que mi idea de hacer 'un Minix mejor que Minix' se hacía más seria. Esperaba que algún día pudiese recompilar el `gcc` bajo Linux..."
>
> "Dos meses de trabajo, hasta que tuve un driver de discos (con numerosos *bugs*, pero que parecía funcionar en mi PC) y un pequeño sistema de ficheros. Aquí tenía ya la versión 0.01 [al final de Agosto de 1991]: no era muy agradable de usar sin el driver de disquetes, y no hacía gran cosa. No pensé que alguien compilaría esa versión."

No se anunció nada sobre esa versión, puesto que las fuentes del 0.01 jamás fueron ejecutables: contenían sólo rudimentos de lo que sería el *kernel*, y se siponía que se tenía acceso a un Minix para poderlo compilar y jugar con él.

El 5 de Octubre de 1991, Linus anunció la primera versión "oficial" de Linux la 0.02. Ya podía ejecutar `bash` (el shell de GNU) y `gcc` (el compilador de C de GNU), pero no hacía mucho más. La intención era ser un juguete para hackers. No había nada sobre soporte a usuarios, distribuciones, documentación ni nada parecido. Hoy, la comunidad de Linux aún trata estos asuntos de forma secundaria. Lo primero sigue siendo el desarrollo del *kernel*.

Linus escribía en `comp.os.minix`,

> "¿Suspirás al recordar aquellos días de Minix-1.1, cuando los hombres eran hombres y escribían sus propios drivers? ¿Te sentís sin ningún proyecto interesante y te gustaría tener un verdadero S.O. que pudieras modificar a placer? ¿Te resulta frustrante el tener sólo a Minix? Entonces, este artículo es para vos:
>
> Como dije hace un mes, estoy trabajando en una versión gratuita de algo parecido a Minix para computadoras At-386. He alcanzado la etapa en la que puede ser utilizable y voy a poner las fuentes para su distribución. Es sólo la versión 0.02... pero he conseguido ejecutar en él `bash`, `gcc`, `gnu-make`, `gnu-sed`, `compress`, etc."

Tras la versión 0.03, Linus saltó a la versión 0.10, al tiempo que más gente empezaba a participar en su desarrollo. Luego de numerosas revisiones, se alcanzó la versión 0.95, reflejando la esperanza de tener lista muy pronto una versión "oficial". (Generalmente, la versión 1.0 de los programas se corresponde con la primera *teóricamente* completa y sin errores). Esto sucedía en Marzo de 1992. Año y medio después, en Diciembre del 93, el *kernel* estaba en la revisión 0.99.pl14, en una aproximación asintótica al 1.0. Actualmente, el *kernel* se encuentra en la versión 1.1 parche 52, y se acerca la 1.2.[3]

[3] N. del T.: En el momento de traducir estas líneas la versión estable del *kernel* es la 2.2.1

Hoy Linux es ya un clónico de UNIX completo, capaz de ejecutar X Window, TCP/IP, Emacs, UUCP y software de correo y News. Mucho software de libre distribución ha sido ya portado a Linux y están empezando a aparecer aplicaciones comerciales. El hardware soportado es mucho mayor que en las primeras versiones del *kernel*. Mucha gente ha ejecutado tests de rendimiento en sus sistemas Linux 486 y se han encontrado que son comparables a las estaciones de trabajo de gama media de Sun Microsystems y Digital. ¿Quién iba a imaginar que este "pequeño" clónico de UNIX iba a convertirse en un estándar mundial para las computadoras personales?

Índice de Materias