CENTRO SUPERIOR DE INFORMÁTICA
Departamento de Estadística, I. O. y Computación

Teoría de Autómatas y Lenguajes Formales

Curso 98/99

CURSO DE INTRODUCCIÓN AL UNIX

http://members.xoom.com/arturovaldes/linuxcur.htm

INTRODUCCION.

1.1 Antecedentes históricos.

El S.O. Unix fue creado a finales de la década de los 60 sobre la base de varios trabajos realizados conjuntamente por el MIT y Laboratorios BELL. Dichos trabajos (proyecto MULTICS) iban encaminados a la creación de un macrosistema de computación que diese servicio a miles de usuarios. Si bien el proyecto fracasó, posiblemente por intentar abarcar demasiado contando con unos elementos hardware limitados en ésa época, influyó; decisivamente sobre la evolución de los sistemas informáticos posteriores.

Un antiguo componente de dicho proyecto (Ken Thompson) desarrolló por su cuenta un sistema operativo monousuario con la característica principal de un sistema de archivos jerárquico.

El sistema encontró muchos entusiastas y se hizo portable al reescribirse casi íntegramente en lenguaje "C", y se suministró en código fuente a las universidades como objeto de formación. Así, la universidad de California en Berkeley retocó dicho sistema (fundamentalmente, comunicaciones y diversas utilidades como el editor "vi"), y liberó lo que luego sería el BSD, uno de los dos "dialectos" principales del UNIX.

Actualmente, existen dos corrientes las cuales cada vez poseen más elementos comunes: la BSD 4.2 y la System V R 4.

1.2 Generalidades.

El S.O. Unix se encarga de controlar y asignar los recursos físicos del ordenador (hardware) y de planificar tareas. Podemos establecer tres elementos principales dentro de este S.O.:

- El núcleo del sistema operativo (kernel), el escalón más bajo que realiza tareas tales como el acceso a los dispositivos (terminales, discos, cintas...).

- El intérprete de comandos (shell) es el interfase básico que ofrece UNIX de cara al usuario. Además de ejecutar otros programas, posee un lenguaje propio así como numerosas características adicionales que se estudiarán en un capítulo posterior.

- Utilidades "de fabrica"; normalmente se trata de programas ejecutables que vienen junto con el Sistema Operativo; algunas de ellas son:

  - Compiladores: C, assembler y en algunos casos Fortran 77 y C++.
  - Herramientas de edición: Editores (vi,ex), formateadores (troff), filtros...
  - Soporte de comunicaciones: Herramientas basadas en TCP/IP (telnet,ftp...)
  - Programas de Administración del Sistema (sysadm, sa, va.....)
  - Utilidades diversas y juegos (este último se suele instalar aparte).

2-ORDENES BASICAS

2.1 Conexión y desconexión.

Para acceder al sistema, este presenta el mensaje de login, con el que quiere significar algo así como "introduce el usuario con el que quieres abrir una sesión".

UNIX(r) System V Release 4.2

login:

Una vez tecleado el usuario que se quiere y haber pulsado RETURN, solicita una palabra de paso (password), la cual, como es natural, no se verá en pantalla aunque se escriba.

```
UNIX(r) System V Release 4.2
login:antonio
Password:
$
```

Tanto el nombre del usuario como la palabra de paso han de ser escritas "de golpe", es decir, no se pueden dar a los cursores para modificar ningún carácter y mucho menos la tecla de Backspace, Ins, Del.... Esto es debido a que, tanto este carácter como los aplicados a los cursores son caracteres válidos en nombres de usuario y en palabras de paso.

El sistema, una vez aceptado el nombre del usuario (el cual como es obvio habrá sido asignado por el Administrador, así como la palabra de paso), lanza por pantalla unos mensajes de bienvenida y termina con el símbolo denominado "prompt", símbolo configurable (como casi todo en UNIX) y que suele ser un '$' ó un '#'.

Existe en todos los sistemas UNIX un superusuario llamado "root", que puede hacer absolutamente lo que quiera en el sistema. Además, hay algunos usuarios especiales, dependiendo del sistema que se trate con más privilegios de los normales (admin ó sa ó sysadm, usuario de administración del equipo, uucp como usuario de comunicaciones) y el resto, que corresponden a usuarios normales.

El programa que está en este momento mostrando el prompt es la shell ó intérprete de comandos. Con este prompt indica algo así como "preparado para que me escribas el comando que quieres ejecutar".

Cada comando debe finalizar en un RETURN, el cual funcionalmente se asemeja a la orden "AR" en la mili (El sargento dice "firmes", pero nadie se mueve hasta que da el RETURN, es decir, "AR").

También es significativa la diferencia entre mayúsculas y minúsculas; no es lo mismo "cal" que "CAL". El primero es un comando de calendario; el segundo no existe y la shell se queja de que no lo encuentra:

```
$ cal
January 1995
S M Tu W Th F S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
$ CAL
UX:sh: ERROR: CAL: Not found  (no existe ningún comando "CAL" !)
```

Para terminar la sesión, se tecleará el comando "exit", lo que provocará la aparición del mensaje de login:

```
$ exit
UNIX(r) System V Release 4.2
login:
```

efecto que puede también conseguirse apagando y encendiendo el terminal (en terminales "civilizados") ó pulsando Ctrl D (Tecla de "Control" y tecla "D" a la vez). Puede que en su terminal alguna de estas dos cosas no funcione: piense que en

UNIX casi todo es configurable.

2.2 Conceptos: grupo,directorio de trabajo, shell, tipo de terminal.

Cada usuario, aparte de la palabra de paso, posee unos determinados atributos los cuales básicamente son:

- Grupo: A cada usuario, en el momento de su creación, se le asigna uno (ó varios, si la versión del sistema es moderno). La misión de esto radica en agrupar usuarios en "grupos" para permitir y denegar accesos en el sistema. Para ver a que grupo pertenecemos, se puede ejecutar el comando "id", el cual producirá un resultado similar al siguiente:

```
$ id
uid=129(jose) gid=1(other)
```

Como era de esperar, el comando nos contesta con siglas en inglés; "uid" (UserID), identificador de usuario, un número y un nombre (129 y jose); "gid" (GroupID), identificador de grupo, un número y un nombre (1 y other).

- Directorio de trabajo: Al iniciar una sesión (tras hacer "login"), cada usuario "cae" en un determinado directorio prefijado. Este habrá sido fijado por el administrador en el momento de la creación, y normalmente estará situado en "/home/jose", "/users/acct/jose" ó similar. No existe un estándar sobre esto y puede ser cualquier subdirectorio del sistema, incluida la raíz.

Para visualizar cual es nuestro directorio, podemos escribir el comando "pwd" (Print Working Directory) inmediatamente después de haber hecho "login".

- Shell: Si bien nos hemos referido de forma genérica a la shell existen diversas versiones asignables a usuarios, de las cuales podemos citar algunas:

- la Bourne shell ("sh"): Una de las más antiguas y, por tanto, más seguras.
- la Korn shell ("ksh"): Basada en la de Bourne pero más moderna y con más funcionalidades.
- la C-Shell ("csh"): Preferida en entornos BSD y bastante potente aunque más criptica que las dos anteriores.

Veremos más adelante las características de cada una de ellas con más detalle.

- Características del Terminal: Dependiendo de cómo se tenga configurado el terminal, la sesión se comporta de una manera determinada frente los siguientes caracteres especiales:

- Control + C (INTR): Interrumpe el programa que se está ejecutando en ese momento.
- Control + H (ERASE): Borra el último carácter escrito.
- Control + D (EOF): Termina la sesión; posee un efecto similar a escribir "exit".

2.3: Comandos (who/date/man/who am i/logname)

En UNIX, un comando se invoca escribiendo su nombre y, separados por blancos, los argumentos opcionales. Como antes, se debe pulsar RETURN una vez se ha escrito la orden correspondiente.

El comando "who" informa de los usuarios que se hallan presentes en el sistema.

```
$ who
jose  ttyp05           Ene 27 10:45
conta1      ttyp15      Ene 27 11:05
carmen      syscons     Ene 27 10:48
```

Si deseamos saber quienes somos (cara al sistema, claro - el unix y la filosofía son cosas distintas), usaremos el comando "who am i".

A veces, no coinciden el usuario con el que uno se ha conectado (con login) y el usuario efectivo. Ya se verá más adelante algo más sobre este tema. En este caso, el comando que muestra con qú e usuario se ha hecho la conexión será "logname".

Los campos corresponden al usuario, terminal asociado y fecha de conexión.

El comando "date" permite ver la fecha y hora del sistema y también fijarla; si escribimos "date" a secas:

```
$ date
Mon Jan 4 18:46:12 GMT 1995
```

Y si intentamos cambiarla, por motivos obvios, solamente podremos hacerlo como el usuario "root" (Las implicaciones que el cambio de fecha trae consigo en relación con los backups incrementales, así como cualquier proceso dependiente de la hora, pueden ser importantes).

No obstante, hay veces que se debe hacer; en estos casos, todos los menús de administración de cualquier equipo UNIX traen el comando "Change Date/Time". No obstante, podemos dar desde "root" el siguiente comando:

```
date -u 1202120094
```

para cambiar a día 02 del mes 12 a las 12.00 del año 1994. (El formato para date es MESMESDIADIAHORAHORAMINUTOMINUTOAÑO).

2.4 Archivos y directorios

Uno de los primeros "manuales" de UNIX (La definición del BSD), establecía que en Unix "todo son ficheros". Tenemos varios tipos:

- Fichero "normal" con sus variantes que más tarde veremos. Es algo que, referido a un nombre, contiene una secuencia determinada de caracteres (bytes). El sistema operativo no impone ningún tipo de formato ni de registro. Ejemplos de ficheros "normales" puede ser el que nos creamos con algún editor, conteniendo un documento. Por ejemplo, el fichero "/etc/motd" es un fichero cuyo contenido es el mensaje del día. Si ejecutamos el comando "cat" (ver el contenido de un fichero), tendríamos una salida como esta:

```
$ cat /etc/motd
***************************************************
© 1981-95 AT&T Unix System Laboratories
Bienvenido al equipo XXXXXXXXXXX
***************************************************
```

El mismo comando "cat" es un fichero, lo que ocurre es que, en vez de contener "letras" legibles, contiene código binario que sólo el equipo es capaz de comprender. Podemos probar esto haciendo:

```
$ cat /bin/cat
kjsdñkqbehls wqej jsdf sdcaj
dfkjsadf df asldfj af ^C ^C ^B aasd (.... BASURILLA....)
```

Pues también son ficheros normales aquellos que contienen programas compilados (ejecutables).

-Directorios, que son ficheros cuyo contenido son nombres de ficheros. Funcionalmente, se comportan de la misma manera que en sistemas tipo MS-DOS.

-Ficheros especiales. Ya que es el propio sistema operativo el que debe ocuparse de los detalles "sucios" de acceso a dispositivos, la apuesta de los creadores y continuadores de UNIX fue el de homogeneizar todos estos accesos vía fichero. De tal forma, si tenemos un fichero correspondiente a una impresora, bastará escribir en él para que salga en papel; de la misma manera, si deseamos ver lo que una cinta Exabyte, DAT ó Streamer contiene debe ser suficiente con "leer" lo que el fichero de acceso a ésos dispositivos contiene.

Si bien la nomenclatura varía entre sistemas, normalmente todos estos tipos residen en el directorio "/dev". Podemos citar algunos de ellos:

```
/dev/lp0,/dev/lp1         : Primera y segunda impresora en paralelo.
/dev/tty00,/dev/tty01....: Terminales ó modems en conexión serie.
/dev/eth0                 : Primera placa ethernet
/dev/dsk/c0d0s0           : Primera partición del disco "0" en la controladora "0"
/dev/rmt/0                : Unidad de cinta Streamer con rebobinado automático.
```

Podemos crear un archivo de múltiples formas; utilizando un editor, mediante un programa ó utilizando algún comando. Unicamente se tendrá en cuenta, a la hora de escoger nombres para los ficheros, las sig uientes reglas:

- El nombre no podrá exceder de 255 letras (14 en versiones UNIX anteriores a la R2.1)

- Utilizar caracteres "normales" (Si bien se puede usar cualquier carácter, desechar aquellos que no sean A-Z /a-z / 0-9) Especial atención merecen aquellos ficheros cuyo primer carácter es el punto ";.", ya que se consideran "ocultos" en la medida en que determinados comandos, al detectar este hecho, los ignoran.

- Vamos a crear un fichero cuyo contenido sea "hola":

```
$ echo "hola" > tonto
$
(... parece que no ha ocurrido nada...)
$ cat tonto
hola
$
(efectivamente - ahora, el comando "ls" nos da un listado de los ficheros)
$ ls
tonto
$ ls -a
.
..
tonto
.profile
$
```

Ahora parece que el fichero "tonto" ha criado! Lo que ocurre es que estamos en nuestro directorio de trabajo, y ahí existe desde el principio un fichero "oculto" (el .profile) el cual sólo podemos listar utilizando la opción "-a" (all) del ls. Además de esto, aparecen el "punto" (.) y el "punto-punto", sinónimos de "directorio actual" y "directorio anterior".

Primero, vamos a ver en qué directorio estamos, para lo cual emplearemos el comando "pwd" (Print Working Directory).

```
$ pwd
/usr/jose
$ ls
tonto
```

Nuestro directorio de trabajo es "/usr/jose" y tenemos dentro del mismo el fichero "tonto". Utilizando los conceptos de "." y "..", podemos ejecutar el comando "ls" pasando los mismos como argumentos:

```
$ ls. (listar el contenido del directorio "actual"; funciona igual que "ls" a secas)
tonto
$ ls..
jose
paco
antonio
bin
(....)
```

Estamos diciendo que nos liste el "directorio anterior", es decir, "/usr".

Podemos hacer lo mismo cambiando al directorio anterior y ejecutando "ls": nos cambiamos de directorio usando el comando "cd" (Change Directory), con direccionamiento absoluto (indicando todo el camino de subdirectorios desde el directorio "raiz" (/):

```
$ cd /usr
$ pwd
/usr
$ ls
jose
paco
antonio
bin
(...)
```

O bien, cambiando con direccionamiento relativo, y se dice así por ser relativo al directorio donde estamos trabajando:

```
$ cd.. (nos cambiamos al directorio anterior, es decir, a "/usr")
$ pwd
/usr
ls
jose
paco
antonio
bin
(...)
```

Para volver a nuestro directorio "origen" (home directory), bastaría con ejecutar el comando "cd" a secas; él vuelve siempre al mismo sitio, en este caso, a "/usr/jose" estemos donde estemos.

Si queremos crear un nuevo directorio emplearemos en camino absoluto ó relativo el comando "mkdir" con el argumento "nombre_del_directorio_que_queremos_crear". Para eliminarlo, ejecutaremos "rmdir" con el mismo argumento:

```
$ mkdir nuevo
$ ls
tonto
nuevo
```

Tenemos ahora el fichero de siempre y un directorio que se llama "nuevo". Pero escribiendo "ls" no vemos a simple vista ninguna diferencia; hay que dar la opción "-F" ó "-l":

```
$ ls -F
tonto
nuevo/
$ ls -l
-rw-r--r--  1 jose      sys   4     Mar 1 11:07 tonto
drw-r--r--  1 jose      sys   84    Mar 1 11:07 nuevo
(la letra "d" al principio del listado largo nos indica que se trata de un
directorio)
$ cd nuevo
$ pwd
/usr/jose/nuevo
(.. nos hemos cambiado al directorio nuevo..)
$ cd..
$ pwd
/usr/jose
(.. homes vuelto...)
$ rmdir nuevo
$ ls
tonto
(.. ! ya no está..)
```

2.4.5 Tipos de archivos.

Si bien UNIX no impone una estructura a ningún fichero, estos tendrán características comunes dependiendo para lo que sirvan; podemos agrupar estos en varios tipos:

- Ejecutables. Normalmente se trata de programas compilados y contienen código binario ininteligible para la mayoría de los humanos pero no así para la máquina. Ejemplos de estos ficheros pueden ser l os ficheros "/bin/ls", "/bin/cat/"... Todos ellos deben tener activados los permisos de ejecución que más tarde veremos.

- Binarios, englobando dentro de esta categoría aquellos que son empleados por programas capaces de entender su contenido (un fichero indexado accesible desde COBOL, por ejemplo) pero no legibles.

-Texto, correspondientes a aquellos ficheros que contienen registros de caracteres terminados en "nueva línea" y normalmente son legibles. Ejemplo de fichero de texto puede ser el "/etc/motd", "/etc/pas swd" y cualquiera que haya sido confeccionado con el editor.

- Dispositivo. Cualquier fichero asignado a un dispositivo físico; normalmente residen a partir del subdirectorio "/dev" y son ficheros de terminales, de particiones de discos, de puertos de impresora.... Dentro de esta categoría podemos incluir el fichero especial "/dev/null", el cual tiene como misión el que cualquier cosa que se le mande es desechada.

Cualquier fichero de tipo dispositivo puede tener dos categorías: bien de tipo "carácter", ó "raw mode" (lee y escribe "de golpe") ó de tipo "bloque" ó "cooked mode" (lee y escribe a trozos según el tamaño de buffer que tenga asignado).

Si se tienen dudas acerca del tipo de fichero, podemos ejecutar el comando "file <nombre_del_fichero>.

```
$ file /bin/ls
/bin/ls: ELF 32-bit MSB executable M88000 Version 1
(Ejecutable dependiente del procesador 88000 de Motorola)
$ file /dev/null
```

character special

2.4.6 Tipos de acceso / permisos.

En cualquier sistema multiusuario es preciso que existan métodos que impidan que un determinado usuario pueda modificar ó borrar un fichero confidencial, ó incluso leer su contenido. Asimismo, determinados comandos (apagar la máquina, por ejemplo) deben estar permitidos exclusivamente a determinados usuarios, quedando inoperantes para los demás.

En UNIX, estos métodos radican en que cada fichero tiene un propietario, que es el usuario que creó el fichero. Además, los usuarios están divididos en grupos, asignación que normalmente se lleva a cabo por el Administrador del sistemas, dependiendo de la afinidad de las tareas que realizan. El fichero anterior tiene también un grupo, que es el grupo del usuario que lo ha creado. UNIX distingue tres tipos de acceso - lectura, escritura y ejecución - sobre el usuario que lo ha creado, los usuarios del mismo grupo que el que lo creó y todos los demás usuarios.

Por todo lo anteriormente dicho, un fichero puede tener cualquier combinación de los tres tipos de acceso sobre tres tipos de usuarios: el creador, los de su grupo y todos los demás (otros cualquiera que no cumplan ninguna de las dos condiciones anteriores). Para ver lso permisos de un fichero cualquiera, empleamos el comando "ls -l" (formato largo):

```
$ ls -l
-rw-r--r--  1 jose      sys   4     Mar 1 11:07 tonto
drw-r--r--  1 jose      sys   84    Mar 1 11:07 nuevo
```

Los campos que lo componen son:

-rw-r--r-- : tipo de fichero (la primera raya; en el caso de "nuevo", una "d" significa que es un directorio, y una raya "-" significa fichero normal) y 3 grupos de tres correspondientes, los tres primeros, a permisos del usuario; los tres siguientes, a los del grupo, y los tres últimos, a todos los demás.

jose : Usuario que creó el fichero.
sys : Grupo del creador.
4 : Tamaño en bytes.
Mar 1 11:07 : Fecha de creación ó de última modificación.
tonto : Nombre del fichero.

Cada grupo de tres elementos pueden ser "rwx", que son: permiso para leer, permiso para escribir y permiso para ejecutar. Una raya significa "carece de permiso para (.)".

Por tanto, el fichero "tonto" puede ser leído y escrito (y, por tanto, borrado), por el usuario "jose". Cualquier otro usuario, sea del grupo "sys" ó no, tiene permisos sólo de lectura.

En el caso de directorios, todo igual salvo que en este caso la "x" de ejecutar no tendría sentido; por ello, aquí este carácter significa "el usuario X puede cambiarse a este directorio".

Para darle ó quitarle permisos a un fichero ó directorio, se emplea el comando chmod <máscara> <nombre_de_fichero>. La máscara es un número octal de hasta un máximo de cuatro cifra s correspondiente a sumar los siguientes números:

1: Permiso de ejecución.
2: Permiso de escritura.
4: Permiso de lectura.

Por tanto, para darle máximos permisos al fichero "tonto", ejecutaremos el comando:

```
$ chmod 777 tonto
$ ls tonto
-rwxrwx     rwx   1 jose      sys   4     Mar 1 11:07 tonto
```

al darle tres sietes (1 + 2 + 4 = 7), el fichero se queda con la máscara rwx.

Si queremos sólo lectura para creador, grupo y otros, el comando sería "chmod 444 tonto" y así sucesivamente.

En determinadas ocasiones puede ser necesario cambiar de propietario a un fichero ó directorio; para ello utilizamos el comando "chown <nombre_del_propietario> <fichero>". Igual pasa con los grupos; el comando es "chgrp <nombre_del_grupo> <fichero>"

```
$ chown juan tonto
$ ls tonto
-rwxrwxrwx  1 juan      sys   4     Mar 1 11:07 tonto
$ chgrp conta tonto
$ ls tonto
-rwxrwxrwx  1 jose      conta 4     Mar 1 11:07 tonto
```

Pero, cuando creamos un fichero, que permisos coge por defecto ? El valor de "umask" que tenemos asignado en la sesión complementado con 666 (rw-rw-rw). El comando "umask" a secas nos devuelve dicho valor: /P

```
$ umask
022
```

En este caso, 666-022 = 644, es decir, cualquier fichero que creemos con un editor ó con otros comandos serán creados con permisos 644 (rw-r--r--).

Para cambiar la máscara, usamos el comando umask con la nueva máscara que le queremos dar:

```
$ umask 000
```

En cuyo caso, todos los ficheros a partir del ése momento, y hasta que finalice la sesión, serán creados con "barra libre" para todo el mundo.

2.4.8 Manipulación de ficheros.

Podemos examinar una primera tanda de comandos hermanos, que son los siguientes:

cp <fichero a copiar> <fichero nuevo>
mv <fichero a mover> <fichero nuevo>
ln <fichero a enlazar> <fichero nuevo>

"cp" copia el primer argumento al segundo. Valen caminos relativos, es decir

```
$ cp tonto /tmp
$ cp tonto /tmp/nuevo
$ cp /home/jose/tonto /tmp/tonto
```

producirían el mismo resultado; copian el fichero desde el directorio actual al /tmp.

El comando "mv" se comporta igual salvo que el fichero original desaparece; es similar al "RENAME" de MS-DOS.

El comando "ln" hace que pueda existir un contenido con varios nombres:

```
$ ln tonto tonto1
$ ls -l tonto*
```

```
-rw-r--r--  2 jose      sys   4     Mar 1 11:07 tonto
-rw-r--r--  2 jose      sys   4     Mar 1 11:07 tonto1
```

ahora "tonto" y "tonto1" hacen referencia al mismo contenido; por tanto, cualquier cambio que realicemos en uno se verá en el otro, y si borramos ahora el fichero "tonto", quedará el " tonto1" con el mismo contenido, que sólo será borrado al borrar el último de los ficheros enlazados a su contenido.

Debido a la naturaleza del propio comando, no se pueden hacer enlaces (links) entre ficheros ó directorios situados en sistemas de ficheros distintos.

Para este último caso, existe una opción del comando "ln", el "ln -s" (link simbólico); crea un enlace simbólico entre dos entidades pero sólo como una referencia; en este caso si el fichero original se borra, el link queda suelto y el contenido irrecuperable.

El comando "rm" borra ficheros y directorios; para borrar el tonto1 que ya no necesitamos, podemos escribir:

```
$ rm tonto1
```

Valen metacaracteres. Si escribimos

```
$ rm *
```

nos borramos TODO lo que haya en el "directorio actual" SIN pedir confirmación.

Una opción de rm lo hace más seguro al pedirla:

```
$ rm -i tonto1
tonto1 ? y
```

La opción más peligrosa puede ser la "-r" (recursivo) que borra ficheros y directorios a partir del directorio de arranque. Por tanto, ojo con poner

```
$ rm -r.
```

si estamos en el directorio "raíz" y tenemos privilegios, podemos borrar el contenido de TODOS LOS DISCOS

Por último, veremos un comando de utilidad similar a "ls" pero con mas opciones; el comando "find" muestra en pantalla directorios y/o ficheros atendiendo a opciones determinadas. Si bien la sintaxis es demasiado extensa para estudiarla en su totalidad, conviene ver algunos ejemplos:

- El comando toma como argumentos declaraciones en formato "find <directorio base> <opciones> -print". Por tanto, para sacar en pantalla todos los ficheros y directorios del disco, valdría con:

```
find / -print
```

- Si se desean sólo ficheros ó sólo directorios:

```
find / -type f -print
ó
find / -type d -print
respectivamente.
```

- Buscar todos los ficheros y directorios que tengan más de 30 días de antigüedad:

```
find / -atime +30 -print
```

El resto de las opciones sirve para buscar ficheros con determinados permisos, mayores ó menores que un determinado tamaño....

2.4.9: Directorios del sistema.

Si bien la estructura de los directorios varía entre versiones y dialectos de UNIX, podemos mencionar algunos de los más representativos:

/bin,/usr/bin,/usr/sbin,/usr/ucb: Directorios de programas ejecutables. Es aquí donde se mantienen la mayoría de los comandos (/bin/ls, /bin/vi, /usr/bin/chmod...)

/etc : Programas y ficheros diversos: aquí residen la mayoría de los archivos de configuración y las shell-scripts de arranque y parada del equipo.

/tmp,/usr/tmp,/var/tmp : Directorios "temporales"; el propio sistema operativo los usa como almacenamiento intermedio en muchas de sus tareas; normalmente, todo el contenido de estos directorios se borra al apagar el equipo.

/home,/usr/acct,/var/acct : Directorios de usuarios.

/var/adm,/usr/adm : Directorios de administración: programas de administrador, ficheros con informaciones de la paquetería instalada y demás.

/usr/log, /var/log : Directorios para ficheros de "log" de sistemas y de programas. Normalmente, existirá un fichero "console" con toda la información que se desvía a la consola del sistema, y varios ficheros informativos, de advertencia y de errores del sistema.

/lib,/usr/lib : Directorios de archivos de librería.

/usr/spool : Directorios de spool de ficheros de impresión (/usr/spool/lp), de comunicaciones (/usr/spool/uucp) y demás.

/usr/mail: En este directorio se halla el correo de todos los usuarios.

/usr/preserve,/var/preserve: Aquí se guardan todas las copias de ficheros incompletamente terminados por los editores de UNIX.

/dev : Directorio de ficheros de dispositivos.

/usr/games : Juegos. Casi nunca están instalados.

Resumen del editor de pantalla "vi".

Vi es un editor de pantalla completa creado en el UNIX de Berkeley y extendido más tarde a todos los demás dialectos.

La ventaja de este editor radica en su compatibilidad con el resto de herramientas de edición UNIX.

Vi contempla, una vez iniciada la sesión con él, dos modos:

- Modo comando: es el modo en el que arranca por defecto. Vale para dar comandos tales como: leer un fichero, escribir un fichero, búsqueda ó sustitución de caracteres dentro del texto actual...

Este modo es al que se vuelve siempre, pulsando la tecla <ESC>.

- Modo inserción/reemplazo: es el que se usa para escribir caracteres. Se entra a este modo desde el modo comando pulsando la letra " i ".

Desde el modo comando, podemos pasar también a modo inserción escribiendo las siguientes letras:

- i: Pasar a modo inserción, delante de la posición del cursor.
- a: Igual, pero detrás de la posición del cursor.
- I: Pasar a modo inserción, pero empezando a insertar al principio de la línea donde se esté.
- A: Igual, pero empezando al final de la línea donde se esté.
- o: Pasar a modo inserción, pero abriendo una línea nueva debajo de donde se esté.
- O: Igual, pero la línea nueva se abre arriba.
- <ESC>: Pasar a modo comando.
- Backspace: Borra la última letra escrita.
- <Control>v: Identifica el carácter que vamos a escribir a continuación como un carácter especial, es decir, un

escape, ó salto de hoja (^L), ó cualquier carácter ascii entre el 1 y el 31. (1=^A, 2=^B.....).

Pasando a modo comando, podemos emplear las siguientes secuencias para movernos por el texto:

- ^: Ir al principio de la línea.
- $: Ir al final de la línea.
- l,h: Izquierda / derecha.
- j,k: Abajo / Arriba.
- ^F (Control + F): Una pantalla adelante.
- ^B (Control + B): Una pantalla atrás.
- ^G (Control +G): Enseña el número de línea donde estamos posicionados.
- 1G: Al principio del todo.
- G: Al final del todo.
- /<cadena>: Busca <cadena> desde la línea actual hasta el final del texto.
- / Sigue buscando más ocurrencias de la cadena.
-?<cadena>: Busca <cadena> desde la línea actual para atrás.
-? Sigue buscando más ocurrencias, para atrás.
-ZZ Grabar y salir.
-dd: Borrar la línea donde se esté.
-J: Juntar la línea donde se esté y la de abajo.
-r: Reemplaza una sola letra.
-R: Reemplaza todo hasta que se pulse <ESC>.
-yy: yank/yank: Marca la línea actual.
-p: Copia la línea marcada después del cursor.
-P: Copia la línea marcada antes del cursor.
-.: Repite el último cambio.
-u Undo (anula el último cambio)
-U Undo, pero en la línea actual.

Ordenes de "dos puntos": Al pulsar en modo comando el carácter ":", el cursor se baja a la última línea y el editor saca los dos puntos en espera de que le demos algún comando de estos. Salimos de aquí pulsando <ESC>.Algunos son:

:wq! (write/quit)Grabar y salir.
:w! <nombre_fichero> (write) Graba el texto en <nombre_fichero>.
:r <nombre_fichero> (read) Incluye el fichero como parte del texto en edición.
:!<comando> Ejecutar <comando>. Vuelve al terminar el comando.
:<número de línea> Ir a <número de línea>.
:q! (quit) Salir sin grabar.
:se nu
:se nonu Numera las líneas.(set number,set nonumber-pone y quita).
:1,5de Borra desde la 1 a la 5.
:1,5co20 Copia desde la 1 a la 5 a partir de la línea 20.
:1,5mo20 Igual, pero mueve.(desaparece desde la línea 1 a la 5).
:g/XXX/s//YYY/g Cambia en todo el texto XXX por YYY.

Hay muchísimos más comandos, pero para ir tirando, con estos son suficientes.

3 LA SHELL

3.1 Introducción.

La shell es el programa más importante para la mayoría de los usuarios y administradores de UNIX; con la excepción del editor de textos y del menú de administración, posiblemente es con el que más tiempo se trabaja.

La shell es el lenguaje de comandos de UNIX; es un programa que lee los caracteres tecleados por los usuarios, los interpreta y los ejecuta.

A diferencia de otros intérpretes más estáticos en otros sistemas operativos, aquí existe además un completo lenguaje de programación que permite adaptar de manera relativamente sencilla el sistema operativo a las necesidades de la instalación.

Una vez que un usuario común se ha registrado cara al sistema con su login y su password, se le cede el control a la shell, la cual normalmente ejecuta dos ficheros de configuración, el general (/etc/profile) y el particular(<directorio_del_usuario>/.profile). Una vez aquí, la propia shell despliega el literal "inductor de comandos", que normalmente será:

$

ó

#

Y el cursor en espera de que alguien escriba algún comando para ejecutar.

Tipos y propiedades.

Ya que, como hemos explicado anteriormente, la shell es un programa, existen varios, cada uno con sus características particulares. Veamos algunas de ellas:

- Bourne shell (/bin/sh): Creada por Steven Bourne de la AT&T. Es la más antigua de todas y, por tanto, la más fiable y compatible entre plataformas. Esta es en la que se basan las explicaciones posteriores.

- Korn shell (/bin/ksh): Creada por David G. Korn de los laboratorios Bell de la AT&T.Más moderna, toma todos los comandos de la Bourne y le añade varios más así como varias características de reedición interactiva de comandos, control de trabajos y mejor rendimiento en términos de velocidad que la anterior. Existen dos versiones, una primera "maldita" y la actual, de 16/11/1988. (podemos averiguar qué versión tiene la nuestra ejecutando el comando "what /bin/ksh") Pero no es oro todo lo que reluce; cuando se trata de situaciones extremas ó complicadas donde la ksh falla, la de Bourne normalmente está más "blindada".

- C Shell, desarrollada por Bill Joy en la Universidad de California y, por tanto, más extendida entre UNIX BSD. Bastante más críptica que la de Bourne, incorpora no obstante sustanciales mejoras.

Estructura de las órdenes - Metacaracteres.

Cuando escribimos cualquier comando y pulsamos <INTRO>, es la shell y no UNIX quien se encarga de interpretar lo que estamos escribiendo y ordenando que se ejecute dicho comando. Aparte de los caracteres normales, la shell inter preta otros caracteres de modo especial: un grupo de caracteres se utiliza para generar nombres de ficheros sin necesidad de teclearlos explícitamente.

Cuando la shell está interpretando un nombre, los caracteres * ? [ ] se utilizan para generar patrones. El nombre que contenga alguno de estos caracteres es reemplazado por una lista de los ficheros del directorio actual cuyo nombre se ajuste al patrón generado.

Las reglas de generación de patrones son:

* Vale cualquier cadena de caracteres.

? Vale un carácter cualquiera.

[..] Vale cualquiera de los caracteres que coincida con los que estén entre corchetes.

Ejemplo. Supongamos que en nuestro directorio actual tenemos los siguientes ficheros:

```
$ ls
tonto
tonta
diario
```

```
mayor
```

Veamos cada una de las salidas correspondientes a las reglas anteriores:

```
$ ls *             (valen todos)
tonto
tonta
diario
mayor
$ ls *o            (todos, pero tienen que acabar en "o")
tonto
diario
$ ls tont?  (que empiecen por tont y cualquier otro carácter)
tonto
tonta
$ ls tont[oa]      (que empiecen por tont y el siguiente sea "o" ó "a")
```

3.2 Concepto de comando / proceso.

Para comprender la manera en la cual la shell ejecuta los comandos hay que tener en cuenta las circunstancias siguientes:

- Tras sacar en pantalla el indicador $, espera a que se le introduzca algo, lo cual será interpretado y ejecutado en el momento de pulsar <INTRO>.

- La shell evalúa lo que hemos escrito buscando primero si contiene un carácter "/" al principio. En caso que sea así, lo toma como un programa y lo ejecuta.

Si no, examina si se trata de una función (o un alias,en el caso de la ksh). Una función es una secuencia de comandos identificada por un nombre unívoco, y se verá más adelante. En caso de no encon trar ninguna con ése nombre, busca a ver si se trata de un comando interno

(exit, exec, trap, etc) ó palabra reservada (case,do,done,if,for.. etc), para ejecutarlo ó pedir más entrada. Si ninguna de estas condiciones es cierta, la shell piensa que lo que hemos escrito es un comando, y lo busca dentro de los directorios contenidos en la variable de entorno PATH. Si no está, saca un mensaje del tipo "XXXX: not found", siendo XXXX lo que hemos escrito.

Mejor verlo con un ejemplo: supongamos que escribimos alguna aberración del tipo:

```
$ hola
```

Suponiendo que la variable PATH contenga los directorios /bin,/usr/bin y /etc, la shell busca el comando "/bin/hola", "/usr/bin/hola" y "/etc/hola". Ya que obviamente no existe, la contestación se rá:

```
sh: hola: not found.
```

La shell utiliza UNIX para la ejecución de procesos, los cuales quedan bajo su control. Podemos definir un proceso aquí como un programa en ejecución. Ya que UNIX es multitarea, utiliza una serie de métodos de "tiempo compartido" en los cuales parece que hay varios programas ejecutándose a la vez, cuando en realidad lo que hay son intervalos de tiempo cedidos a cada uno de ellos según un complejo esquema de prioridades.

Cuando la shell lanza un programa, se crea un nuevo proceso en UNIX y se le asigna un número entero (PID) entreel 1 y el 30.000, del cual se tiene la seguridad que va a ser unívoco mientras dure la sesión. Lo podemos ver ejecutando el comando "ps", el cual nos da los procesos activos que tenemos asociados a nuestro terminal.

Un proceso que crea otro se le denomina proceso padre. El nuevo proceso, en este ámbito, se le denomina proceso hijo. Este hereda casi la totalidad del entorno de su padre (variables, etc)., pero sólo puede modificar su entorno, y no el del padre.

La mayoría de las veces, un proceso padre se queda en espera de que el hijo termine; esto es lo que sucede cuando lanzamos un comando; el proceso padre es la shell, que lanza un proceso hijo (el comando). Cuando este comando acaba, el padre vuelve a tomar el control, y recibe un número entero donde recoge el código de retorno del hijo (0=terminación sin errores, otro valor=aquí ha pasado algo).

Cada proceso posee también un número de "grupo de procesos". Procesos con el mismo número forman un sólo grupo, y cada terminal conectado en el sistema posee un sólo grupo de procesos. (Coma ndo ps -j). Si uno de nuestros procesos no se halla en el grupo asociado al terminal, recibe el nombre de proceso en background (segundo plano).

Podemos utilizar algunas variantes del comando "ps" para ver qué procesos tenemos en el equipo:

ps : muestra el número de proceso (PID), el terminal, el tiempo en ejecución y el comando. Sólo informa de nuestra sesión.

ps -e : de todas las sesiones.

ps -f : full listing: da los números del PID,del PPID(padre),uso del procesador y tiempo de comienzo.

ps -j : da el PGID (número de grupo de los procesos - coincide normalmente con el padre de todos ellos)

Este comando puede servirnos para matar ó anular procesos indeseados. Se debe tener en cuenta que cada proceso lleva su usuario y por tanto sólo el (ó el superusuario) pueden matarlo.

Normalmente, si los programas que componen el grupo de procesos son civilizados, al morir el padre mueren todos ellos siempre y cuando el padre haya sido "señalizado" adecuadamente. Para ello, empleamos el comando kill -<número de señal> PID, siendo PID el número del proceso ó del grupo de procesos.

Los números de señal son:

-15 : TERM ó terminación. Se manda para que el proceso cancele ordenadamente todos sus recursos y termine.

-1 : corte

-2 : interrupción.

-3 : quit

-5 : hangup

-9 : kill: la más enérgica de todas pero no permite que los procesos mueran ordenadamente.

3.3 Entrada y salida. Interconexión.

Para cada sesión, UNIX abre tres ficheros predeterminados, la entrada estándar, la salida estámdar y el error estándar, como ficheros con número 0,1 y 2. La entrada es de donde un comando obtiene su información de entrada; por defecto se halla asociada al teclado del terminal. La silida estándar es donde un comando envía su resultado; por defecto se halla asociada a la pantalla del terminal; el error estándar coincide con la salida estándar, es decir,con la pantalla.

A efectos de modificar este comportamiento para cualquier fin que nos convenga, la shell emplea 4 tipos de redireccionamiento:

< Acepta la entrada desde un fichero.

> Envía la salida estándar a un fichero.

>> Añade la salida a un archivo existente. Si no existe, se crea.

| Conecta la salida estándar de un programa con la entrada estándar de otro.

Intentar comprender estas explicaciones crípticas puede resultar cuanto menos, confuso; es mejor pensar en términos de "letras":

< En vez de coger líneas del teclado, las coge de un fichero. Mejor, adquirir unos cuantos conceptos más para entender

u ejemplo.

> Las letras que salen de un comando van a un fichero. Supongamos que ejecutamos el comando ls usando esto:

```
$ ls > cosa
$     (.. parece que no ha pasado nada...)
$ ls
tonto
tonta
diario
mayor
cosa
$     (.. se ha creado un fichero que antes no estaba - cosa...)
$ cat cosa
tonto
tonta
diario
mayor
cosa
$     (.. y contiene la salida del comando "ls"...)
```

>> En vez de crear siempre de nuevo el fichero, añade las letras al final del mismo.

| El comando a la derecha toma su entrada del comando de la izquierda. El comando de la derecha debe estar programado para leer de su entrada; no valen todos.

Por ejemplo, ejecutamos el programa "cal" que saca unos calendarios muy aparentes. Para imprimir la salida del cal, y tener a mano un calendario sobre papel, podemos ejecutar los siguientes comandos:

```
$ cal 1996 > /tmp/cosa        (el fichero "cosa" contiene la salida del "cal")
$ lp /tmp/cosa                (sacamos el fichero por impresora).
```

Hemos usado un fichero intermedio. Pero, ya que el comando "lp" está programado para leer su entrada, podemos hacer (más cómodo, y ahorramos un fichero intermedio):

```
$ cal 1996 | lp
```

El uso de la interconexión (pipa) exige que el comando de la izquierda lance información a la salida estándar. Por tanto, podríamos usar muchos comandos, tales como cat, echo, cal, banner, ls, find, wh o..... Pero, el comando de la derecha debe estar preparado para recibir información por su entrada; no podemos usar cualquier cosa. Por ejemplo, sería erróneo un comando como:

ls | vi

(si bien ls si saca caracteres por su salida estándar, el comando "vi" no está preparado para recibirlos en su entrada).

Y la interconexión no está limitada a dos comandos; se pueden poner varios, tal que así:

```
cat /etc/passwd | grep -v root | cut -d":" -f2- | fold -80 | lp
```

(del fichero /etc/passwd sacar por impresora aquellas líneas que no contengan root desde el segundo campo hasta el final con un ancho máximo de 80 columnas).

Normalmente, cualquier comando civilizado de UNIX devuelve un valor de retorno (tipo ERRORLEVEL del DOS) indicando si ha terminado correctamente ó bien si ha habido algún error; en el caso de comandos compuestos como e s este, el valor global de retorno lo da el último comando de la pipa; en el último ejemplo, el retorno de toda la línea sería el del comando "lp".

3.5 Variables de entorno.

Una variable de entorno en la shell es una referencia a un valor. Se distinguen dos tipos: locales y globales.

Una variable local es aquella que se define en el shell actual y sólo se conocerá en ese shell durante la sesión de conexión vigente.

Una variable global es aquella que se exporta desde un proceso activo a todos los procesos hijos.

Para crear una variable local:

cosa="ANTONIO ROMERO"

Para hacerla global

export cosa

Para ver qué variables tenemos:

```
# set
LOGNAME=root
TERM=vt220
PS1=#
SHELL=/bin/sh
(.. salen mas..)
```

Una variable se inicializa con la expresión <variable>=<valor>. Es imprescindible que el carácter de igual '=' vaya SIN espacios. Son lícitas las siguientes declaraciones:

```
# TERM=vt220
# TERM="vt220"
(TERM toma el mismo valor en ambos casos)
# contador=0
```

Una variable se referencia anteponiendo a su nombre el signo dòlar '$'. Utilizando el comando 'echo', que imprime por la salida estándar el valor que se le indique, podemos ver el contenido de algunas de estas va riables:

```
# echo TERM
TERM
(!! MUY MAL!! - Hemos dicho que la variable se referencia por $).
# echo $TERM
vt220
(AHORA SI)
```

Otra metedura de pata que comete el neófito trabajando con variables:

```
# $TERM=vt220
```

Y el señor se queda tan ancho. Investigando qué és esto, la shell interpreta que le queremos poner algo así como "vt220=vt220", lo cual es erróneo.

Para borrar una variable, empleamos el comando "unset" junto con el nombre de la variable que queremos quitar, como:

```
# echo $cosa                  (saca el valor de dicha variable)
ANTONIO ROMERO          (el que le dimos antes)
# unset cosa                  (BORRAR la variable)
# echo $cosa
#                       (ya no tiene nada)
```

Cuidadito con borrar variables empleadas por programas nativos de UNIX, tales como TERM ! Si borráis esta variable, el editor "vi" automáticamente dejará de funcionar.

Otro problema que es susceptible de suceder es el siguiente: supongamos una variable denominada COSA y otra denominada

COSAS. La shell, en el momento de evaluar la expresión "$COSAS", se encuentra ante la siguiente disyuntiva:

- Evaluar $COSA y pegar su contenido a la "S" (<contenido de COSA> + "S")

- Evaluar $COSAS, empleando intuición.

Cara a la galería, ambas evaluaciones por parte de la shell serían correctas, pero dependiendo de lo que nosotros queramos hacer puede producir efectos indeseados. A tal fin, en conveniente utilizar los caracteres ";llave" -{}- para encerrar la variable que queremos expandir. De tal forma, para reflejar "COSA", escribiríamos:

${COSA}

Y para reflejar "COSAS",

${COSAS}

Con lo que tenemos la seguridad de que las variables siempre son bien interpretadas. Las llaves se utilizan SIEMPRE en el momento de evaluar la variable, no de asignarle valores. No tiene sentido hacer cosas como {COSAS}=tonterí;a.

Algunas de las variables usadas por el sistema ó por programas del mismo son:

HOME:: Directorio personal. Usado por el comando "cd", se cambia aquí al ser llamado sin argumentos.

LOGNAME: Nombre del usuario con el que se ha comenzado la sesión.

PATH: Lista de rutas de acceso, separadas por dos puntos ':' y donde una entrada con un sólo punto identifica el "direcorio actual". Son válidas asignaciones como:

PATH=$PATH:/home/pepe:/home/antonio

PS1: Símbolo principal del indicador de "preparado" del sistema. Normalmente, su valor será '#' -o '$'.

TERM: Tipo de terminal.

Podemos ver cómo se inicializan las variables consultando los ficheros de inicialización. Estos ficheros son:

/etc/profile: Fichero de inicialización global. Significa que, tras hacer login, todos los usuarios pasan a través del mismo. Inicializa variables como PATH, TERM....

<directorio usuario>/.profile: Fichero particular, reside en el "home directory" del usuario en cuestión. Es, por panto, particular para cada uno de ellos y es aquí donde podemos configurar cosas tale s como que les salga un menú al entrar, mostrarles el correo...

3.6 Sustitución de comandos: acción de las comillas en la shell.

Dependiendo de cuales sean las comillas utilizadas en una expresión, los resultados son los siguientes:

- Carácter backslash \ :Quita el significado especial del carácter a continuación de la barra invertida.

- Comillas simples ' ' :Quitan el significado especial de todos los caracteres encerrados entre comillas simples.

- Comillas dobles " " :Quitan el significado especial de todos los caracteres EXCEPTO los siguientes: $ (dolar), \ (backslash) y ` (comilla de ejecución).

- Comillas de ejecución ` `:Ejecutan el comando encerrado entre las mismas y sustituyen su valor por la salida estándar del comando que se ha ejecutado.

Es mejor, sobre todo en el último caso, ver algunos ejemplos:

- Para sacar un cartel en pantalla que contenga comillas, deberemos "escaparlas" pues, si no, la shell las interpretaría, como en:

echo "Pulse "INTRO" para seguir" (MAL!! - la shell ve 4 comillas y no las sacaría !)

echo "Pulse \"INTRO\" para seguir" (AHORA SI sale bien)

- También, podríamos haber escrito el texto entre comillas simples:

echo 'Pulse "INTRO" para seguir' (BIEN como antes)

lo que ocurre es que de esta manera no se interpretaría nada; nos podría convenir algo como:

echo 'Oye, $LOGNAME pulsa "INTRO" para seguir'

y saldría:

Oye, $LOGNAME pulsa INTRO para seguir

Lo cual no vale. Habría que poner:

echo "Oye, $LOGNAME pulsa \"INTRO\" para seguir"

y saldría:

Oye, root pulsa INTRO para seguir.

- En el caso de comillas de ejecución, podemos escribir:

echo "Oye, `logname` pulsa \"INTRO\" para seguir"

(sale bien, la shell sustituye el comando logname por su resultado)

o bien, valdrían expresiones como:

echo "Y estas en el terminal: `tty`"

Y estas en el terminal /dev/ttyp002

Hay que imaginarse, por tanto, que la shell "ve" el resultado del comando en la línea de ejecución.

Valen también, como es lógico, asignaciones a variables:

TERMINAL=`tty`

echo $TERMINAL

/dev/ttyp001

3.7 Programación con shell.scripts.

La shell, además de interpretar y ejecutar comandos, tiene primitivas de control de ejecución de programas tales como sentencias condicionales y bucles.

La interpretación del lenguaje se lleva a cabo prácticamente en tiempo real; a medida que va interpretando va ejecutando.

Los programas, como se ha mencionado antes, se interpretan en tiempo de ejecución. Por tanto, la codificación de una shell-script es sumamente sencilla en el sentido en el que basta con escribir en un fichero de texto l os comandos y ejecutarlo.

- Variables.

Dentro de una shell, existen las variables de entorno que hayamos definido anteriormente, bien en la misma, en otra ó en los ficheros profile de inicialización. Además de estas, existen otras que detallam os a continuación:

$0 : Su contenido es el nombre de la shell-script que estamos ejecutando.

$1, $2 : Primer y segundo parámetro posicional.

$# : Número de parámetros que han pasado a la shell.

$* : Un argumento que contiene todos los parámetros que se han pasado ($1, $2...) menos el $0.

$? :Número donde se almacena el código de error del último comando que se ha ejecutado.

$$:Número de proceso actual (PID)

$! :Ultimo número de proceso ejecutado.

:COMENTARIO: Todo lo que haya a la derecha de la almohadilla se toma como comentario.

Ejemplo: Supongamos que hemos escrito la siguiente shell-script llamada "prueba.sh":

echo "La escript se llama $0"

echo "Me han llamado con $# argumentos"

echo "El primero es $1"

echo "Y todos son $*"

echo "Hasta luego lucas!"

Y la podemos ejecutar de dos maneras:

1) Directamente:

```
# sh prueba.sh uno dos
```

2) Dando permisos y ejecutando como un comando:

```
# chmod 777 prueba.sh
# prueba.sh uno dos
```

La salida:

```
Me han llamado com 2 argumentos
El primero es uno
Y todos son uno dos
Hasta luego lucas
```

Hemos visto que los comandos se separan por líneas, y se van ejecutando de forma secuencial. Podemos, no obstante, poner varios comandos en la misma línea separandolos por punto y coma ';'.

Además, podemos agrupar comandos mediante paréntesis, lo cual permite ejecutarlos en un subentorno (las variables que usemos no nos van a interferir en nuestro proceso "padre")

```
# (date; who) | wc -l
```

Normalmente, ejecutar una shell implica crear un proceso hijo, y el proceso padre (normalmente, nuestra sesión inicial de shell) espera a que dicho proceso acabe para continuar su ejecución (si nosotros lanzamos un pro grama shell-script, hasta que este no acaba (hijo), no nos aparece en pantalla el inductor de comandos '#' (padre)).

Por definición, en UNIX un proceso hijo, al rodar en un espacio de datos distinto, hereda varias cosas del padre, entre ellas todas las variables de entorno; pero por ello, no puede modificar el entorno del padre (si modif icamos en una shell script el contenido, por ejemplo, de "TERM", al acabar dicha shell y volver al padre la variable continúa con su valor original. Hay situaciones en las cuales necesitamos que una shell modifique nuestro entorno actu al, y a tal efecto se ejecuta con un punto (.) delante de la shell-script.

Es mejor ver este último aspecto mediante un programa en shell.script: supongamos una shell que se llame "tipoterm" que nos pida el terminal y nos ponga la variable TERM de acuerdo a esta entrada:

```
# script para pedir el tipo de terminal
echo "Por favor escriba que terminal tiene:"
read TERM
echo "Ha elegido --- $TERM"
```

Si la ejecutamos como

```
# tipoterm
```

al volver al '#' NO se ha modificado la variable! Para que SI la modifique, se llamaría como:

```
#. tipoterm
```

Hay que suponerse al punto como un "include", que en vez de crear un proceso hijo "expande" el código dentro de nuestra shell actual.

- Comando read

Con el fin de permitir una ejecución interactiva, existe el comando "read <nombre_variable>", el cual, en el momento de ejecución, espera una entrada de datos por teclado terminada en <INTRO>; lo que han introducido por el teclado va a la variable especificada.

Supongamos la siguiente shell-script:

```
echo "Como te llamas?"
read nom
echo "Hola $nom"
```

Ejecución:

```
Como te llamas ?
jose        (aquí escribimos "jose" y pulsamos <INTRO>)
Hola jose
```

el comando "read", ha cargado "jose" en "nom".

- Secuencias condicionales: if.. fi:

La sintaxis de esta sentencia es:

```
if <condicion>
then
..... comandos....
else
..... comandos....
fi
```

(la cláusula "else" puede omitirse; sólo se usará cuando se requiera).

La condición puede escribirse como "test <argumentos>" ó con corchetes. Es imprescindible en este último caso, poner espacios entre los corchetes y los valores.

Posibles condiciones y su sintaxis:

if [<variable> = <valor>] : variable es igual a valor. Ojo con los espacios en '='.

if [<variable> != <valor>] : variable es distinta a valor.
if [<variable -eq <valor>]: variable es igual a valor. La variable debe contener números. En este caso, valen las comparaciones siguientes:
-eq : Igual (equal)
-ne : Distinto (not equal)
-ge : Mayor ó igual (Greater or equal).
-le : Menor ó igual (Less or equal).
-lt : Menor que (Less than).
-gt : Mayor que (Greater than).
if [-f <fichero>] : Existe <fichero>. Ojo con los espacios.
if [-d <fichero>] : Existe <fichero> y es un directorio.
if [-s <fichero>] :Existe <fichero> y tiene un tamaño mayor de cero.
if [-x <fichero>] : Existe <fichero> y es ejecutable.

(Hay mas, pero con estos de momento es suficiente).

En el campo <condición> vale escribir comandos, los cuales se ejecutarán y el valor de la condición dependerá de dos factores:

* Retorno 0 del comando = VERDADERO.

* Retorno != 0 del comando = FALSO.

Ejemplo de esto último sería el siguiente programa:

```
if grep jose /etc/passwd
then  # retorno del comando -grep- ha sido CERO
      echo "Jose esta registrado como usuario"
else  # retorno del comando grep NO ha sido CERO.
      echo "Jose NO esta registrado como usuario"
fi
```

- Secuencia condicional case.. esac.

Sintaxis:

```
case <variable> in
<valor>) <comando>            (la variable es = valor, ejecuta los comandos hasta
';;')
       <comando>
;;
<valor>) <comando>
<comando>
      ;;
*     ) <comando>       (Clausula "otherwise" ó "default": Si no se cumple alguna
<comando>         de las anteriores ejecuta los comandos hasta ';;')
;;
esac                    (Igual que if acaba en fi, case acaba en esac)
```

Ejemplos: minimenu.sh

```
clear                   # borrar pantalla
echo "1.- Quien hay por ahi ?"      # pintar opciones
echo "2.- Cuanto disco queda ?"
echo "3.- Nada. Salir. "
echo "Cual quieres ?: \c"     # el carácter "\c" evita que el echo salte nueva
línea
read opcion             # "opcion" vale lo que se ha tecleado en pantalla
case "$opcion" in       # IMPORTANTE: Poner la variable como "$opcion"
1)    who;;             # pues si el señor pulsa <INTRO> daría error al no valer
```

```
nada.
2)    df;;
3)    echo "Adios";;
*)    echo "Opcion $opcion Es Incorrecta";;
esac
```

- Bucles FOR.

Sintaxis:

```
for <variable> in <lista>
do
<.. comandos..>
done
```

El bloque entre "for" y "done" da tantas vueltas como elementos existan en <lista>, tomando la variable cada uno de los elementos de <lista> para cada iteración. En esto conviene no confun dirlo con los for..next existentes en los lenguajes de tipo algol (pascual, basic...) que varían contadores.

Supongamos un programa que contenga un bucle for de la siguiente manera:

```
      for j in rosa antonio
      do
      echo "Variable = $j"
      done
```

Y la salida que produce es:

```
Variable es rosa
Variable es antonio
```

Explicación: el bloque ha efectuado dos iteraciones (dos vueltas). Para la primera, la variable -j- toma el valor del primer elemento -rosa-, y para la segunda, -antonio-.

En el campo <lista> podemos sustituir la lista por patrones de ficheros para la shell, la cual expande dichos patrones por los ficheros correspondientes; de tal forma que al escribir

for j in *

la shell cambia el '*' por todos los ficheros del directorio actual. Por tanto, el siguiente programa:

```
      for j in *
      do
      echo $j
      done
```

equivale al comando 'ls' sin opciones - merece la pena detenerse un momento para comprender esto.

Vale tambien poner en el campo <lista> comillas de ejecución junto con cualquier comando; la construcción - for j in `cat /etc/passwd` -, por ejemplo, ejecutaría tantas iteraciones como líneas tuviese dicho fichero, y para cada vuelta, la variable -j- contendría cada una de las líneas del mismo. Por tanto, valdrían expresiones como - for j in `who` - para procesar todos los usuarios activos en el sistema, - for j in `lpstat -o ` -, para procesar todos los listados pendientes, ó - for j in `ps -e` - para tratar todos los procesos de nuestra sesión.

- Bucles WHILE.

Sintaxis:

```
while <condición>
do
(... comandos...)
done
```

Aquí las iteraciones se producen mientras que la <condición> sea verdadera (ó retorno = 0). En caso que sea falsa (ó retorno != 0), el bucle termina.

La sintaxis de <condición> es igual que en el comando -if-.

Ejemplo:

```
while [ "$opcion" != "3" ]
do
      echo "Meta opcion"
      read opcion
done
```

ó también, utilizando comandos:

```
echo "Escribe cosas y pulsa ^D para terminar"
while read cosa
do
echo $cosa >> /tmp/borrame
done
echo "Las líneas que has escrito son:"
cat /tmp/borrame
```

Explicación: El comando -read- devuelve un retorno VERDADERO (cero) mientras que no se pulse el carácter EOF (^D); por tanto, el bucle está indefinidamente dando vueltas hasta dar ése carácter.

- Contadores: sentencia expr.

La sentencia expr evalúa una expresión y la muestra en la salida estándar. La expresión normalmente consiste de dos números ó variables de contenido numérico y un operador de suma, resta, multiplicación ó división.

Son válidos los comandos siguientes:

```
expr 100 "+" 1     # saca en pantalla 101
expr 100 "-" 1     # saca en pantalla 99
expr 100 "*" 2     # OJO CON LAS COMILLAS DOBLES- Previenen a la shell de sustituciones.
                   # Bueno, todo el mundo sabe lo que es 100 * 2, no?.
expr 100 "/" 2
```

Por tanto, podemos escribir:

```
pepe=`expr 10 "*" 5`     # y la variable pepe vale 50.
```

ó incluso:

```
pepe=0
pepe=`expr $pepe "+" 1`
```

Esto último es bastante menos evidente. Para comprenderlo, hay que creerse que la shell ejecuta lo siguiente:

- Al principio, $pepe vale 0.

- En cualquier expresión de asignación, PRIMERO se calcula el resultado y DESPUES se ejecuta la asignación. Por tanto, lo primero que la shell hace es "expr 0 + 1".

- El "1" resultante va a sacar por la salida estándar. Pero como hemos puesto las comillas de ejecución, se asigna a pepe.

- Al final, $pepe vale 1.

Pues ya se pueden ejecutar bucles con contadores. Considerese el siguiente programa:

```
cnt=0
while [ $cnt -lt 50 ]
do
```

```
        cnt=‘expr $cnt "+" 1‘
        echo "Vuelta numero $cnt"
done
```

Se autoexplica.

- Operadores AND / OR.

Una construcción usual en la shell, utilizada principalmente por lo compacto de su código, pero con el inconveniente de que permite oscurecer el código son el operador "OR" -||- y el operador "AND" -&&-.

El operador "OR" ejecuta el primer comando, y si el código de error del mismo es FALSO (distinto de cero), ejecuta el segundo comando.

El operador "AND" ejecuta el primer comando, y si el código de error del mismo es VERDADERO (igual a cero), ejecuta el segundo comando.

Veamos un ejemplo y, por favor, reléase a continuación los dos párrafos anteriores:

cd /home/basura && rm -f *

Explicación: nos cambiamos al directorio indicado. Sólamente en el caso de haber tenido éxito, nos cargamos todos los ficheros del directorio actual.

ls /home/basurilla || mkdir /home/basurilla

Explicación: El comando ls falla si no existe el directorio indicado. En tal caso, se crea.

banner "hola" | lp && echo "Listado Mandado" || echo "Listado ha cascado"

Explicación: El comando son dos, el banner y el lp. Si por lo que sea no se puede imprimir, da el mensaje de error; si va todo bien, sale Listado Mandado.

- Depuración de "shell scripts".

Si bien los métodos utilizados para esto son bastante "toscos", ha de tenerse en cuenta que la shell NO se hizo como un lenguaje de programación de propósito general. Cuando se requieren virguerías, ha de acudirse bien a lenguajes convencionales ó bien a intérpretes más modernos y sofisticados (y más complicados, por supuesto), tales como el TCL (Task Control Language) ó el PERL, los cuales si bien son de libre dominio no vienen "de fabrica" en todos los equipos.

Normalmente, emplearemos el comando "set", el cual modifica algunos de los comportamientos de la shell a la hora de interpretar los comandos:

set -v : Verbose. Saca en la salida de error su entrada (es decir, las líneas del script según los va leyendo, que no ejecutando, pues pirmero se lee y después se ejecuta, que esto es un intérpr ete y no hay que olvidarlo).

set -x :Xtrace. Muestra cada comando según lo va ejecutando por la salida de error, antecedido por un "+".

set -n :Noexec. Lee comandos, los interpreta pero NO los ejecuta. Vale para ver errores de sintaxis antes de probarlo de verdad.

set -e :Errexit. Terminar la ejecución inmediatamente si alguno de los comandos empleados devuelve un retorno distinto de VERDADERO (0) y NO se evalúa su retorno. El retorno de un comando se determina evaluado en las siguientes sentencias:

if..fi, while do..done, until do..done.

a la izquierda del operador AND/OR (-||- ó -&&-).

- Comandos trap/exec/exit.

Como se vió en un capítulo anterior, cualquier proceso UNIX es susceptible de recibir señales a lo largo de su tiempo de ejecución. Por ejemplo, si el sistema se apaga (shutdown), todos los procesos recibe n de entrada una señal 15 (SIGTERM), y, al rato, una señal 9.Por ahora, recordaremos que sólamente la señal 9 (SIGKILL) no puede ser ignorada ni redirigida. Un programa puede cambiar el tratamiento que los procesos hacen respecto de las señales, que suele ser terminar.

Las shell-scripts pueden efectuar cosas dependiendo de la señal que reciban, usando el comando "trap <comandos> <señales>". Este comando se suele poner al principio de la shell para que siga vigente a lo largo de toda la ejecución.

Así, podemos directamente ignorar una señal escribiendo lo siguiente:

trap "" 15

(Si llega la señal 15 (SIGTERM), no hagas nada)

O evitar que nos pulsen Control-C, y si lo hacen, se acabó:

trap ‘echo "Hasta luego lucas"; exit 1’ 2 3 15

(Si llega cualquiera, sacar el mensajito y terminar la ejecución con código de retorno 1).

Ojo con la ejecución de subshells - las señales ignoradas (las del trap ‘’) se heredan pero las demás vuelven a su acción original.

El comando ‘exec <comando>’, aparte de una serie de lindezas sobre redireccionamientos cuyo ámbito queda fuera de este manual, reemplaza el proceso de la shell en cuestión con el del comando, el cual debe s er un programa ó otra shell-script. Las implicaciones que esto tiene son que no se vuelve de dicha ejecución. Veamoslo, para no perder la costumbre, con un ejemplo:

Caso 1:

echo "Ejecuto el comando ls"

ls

echo "Estamos de vuelta"

Caso 2:

echo "Ejecuto el comando ls"

exec ls

echo "Estamos de vuelta"

En el caso -1-, la shell actual ejecuta un hijo que es el comando "ls", espera a que termine y vuelve, es decir, sigue ejecutando el echo "Estamos de vuelta". Sin embargo, en el caso -2- esto no es así; hacer exec implica que el número de proceso de nuestra shell pasa a ser el del comando "ls", con lo que no hay regreso posible, y por tanto, el echo "estamos de vuelta", NUNCA podría ejecutarse.

Por tanto, al ejecutar un nuevo programa desde exec, el número de proceso (PID) no varía al pasar de un proceso a otro.

-Comando exit <código de retorno>.

Como ya hemos visto parcialmente, este comando efectúa dos acciones:

Termina de inmediato la ejecución del shell-script, regresando al programa padre (que lógicamente podría ser otra shell ó directamente el inductor de comandos).

Devuelve al proceso antes citado un código de retorno, que podremos averiguar mirando la variable -$?-.

Al terminar una shell script, el proceso inmediatamente antes de acabar, recibe una señal 0, útil en ocasiones para ver por dónde hemos salido usando el comando "trap".

- Funciones.

De manera similar a las utilizadas en lenguajes convencionales, dentro de una shell se pueden especificar funciones y pasarle parámetros. La sintaxis de una función sería:

nombre_funcion()

{

... comandos...

}

Las variables de entorno de la función son las mismas que las de la propia shell-script; debemos imaginar que la propia función se halla "incluída" en la porción de código desde donde la inv ocamos. Por tanto, una variable definida en una función queda definida en la shell y viceversa.

La excepción se produce con los parámetros posiciones; el $1, $2.... cambian su sentido dentro de las funciones, en las cuales representan los parámetros con los que se la ha llamado.

Veamos para variar un ejemplo:

programa de prueba - llamar con parametros

echo "Hola $1"

pinta()

{

echo "Hola en la función $1"

}

pinta "gorgorito"

Veamos el resultado de la ejecución:

sh prueba "probando"

Hola probando

Hola en la funcion gorgorito

La variable -$1-, dentro del programa toma el valor del primer parámetro (probando), y dentro de la función, el parámetro que se le ha pasado a la función (gorgorito). Una vez la función termina, e l $1 vale lo mismo que al principio.

La función nos puede devolver códigos de retorno utilizando la cláusula "return <codigo de error>".

- Ejecución en segundo plano: &, wait y nohup.

Si un comando en la shell termina en un umpresand -&-, la shell ejecuta el comando de manera asíncrona, es decir, no espera a que el comando termine.

La sintaxis para ejecutar un comando en segundo plano es:

comando &

Y la shell muestra un numerito por la pantalla, indicativo del número del proceso que se la lanzado. Hay que tener en cuenta que ése proceso es hijo del grupo de procesos actual asociado a nuestro terminal; significa que si apagamos el terminal ó terminamos la sesión, tal proceso se cortará.

Para esperar a los procesos en segundo plano, empleamos el comando "wait" que hace que la shell espere a que todos sus procesos secundarios terminen.

Siguiendo con lo de antes, hay veces que ejecutamos:

comando_lentisimo_y_pesadisimo &

Y queremos apagar el terminal e irnos a casa a dormir; a tal efecto, existe el comando "nohup" (traducción: no cuelgues, es decir, aunque hagas exit ó apagues el terminal, sigue), que independiza el proces o en segundo plano del grupo de procesos actual con terminal asociado.

Lo que esto último quiere decir es que UNIX se encarga de ejecutar en otro plano el comando y nosotros quedamos libres de hacer lo que queramos.

Una duda: al ejecutar un comando en background, la salida del programa nos sigue apareciendo en nuestra pantalla, pero si el comando nohup lo independiza, que pasa con la salida ? La respuesta es que dicho comando crea un fichero ll amado "nohup.out" en el directorio desde donde se ha lanzado, que coniene toda la salida, tanto normal como de error del comando.

Ejemplo sobre cómo lanzar comando_lentisimo_y_pesadisimo:

nohup comando_lentisimo_y_pesadisimo &

Sending output to nohup.out

12345

#

El PID es 12345, y la salida del comando la tendremos en el fichero "nohup.out", el cual es acumlativo; si lanzamos dos veces el nohup, tendremos dos salidas en el fichero.

- Comandos a ejecutar en diferido: at, batch y cron.

Estos tres comandos ejecutan comandos en diferido con las siguientes diferencias:

AT	lanza comandos una sola vez a una determinada hora.
BATCH	lanza comandos una sola vez en el momento de llamarlo.
CRON	lanza comandos varias veces a unas determinadas horas, días ó meses.

Estos comandos conviene tenerlos muy en cuenta fundamentalmente cuando es necesario ejecutar regularmente tareas de administración ó de operación. Ejemplos de situaciones donde estos comandos nos pueden ayuda r son:

- Necesitamos hacer una salva en cinta de determinados ficheros todos los días a las diez de la mañana y los viernes una total a las 10 de la noche = CRON.

- Necesitamos dejar rodando hoy una reconstrucción de ficheros y apagar la máquina cuando termine (sobre las 3 de la mañana), pero nos queremos ir a casa (son ya las 8) =AT

- Necesitamos lanzar una cadena de actualización, pero estan todos los usuarios sacando listados a la vez y la máquina está tumbada = BATCH

-Comando at <cuando> <comando a ejecutar>

Ejecuta, a la hora determinada, el <comando>. Puede ser una shell-script, ó un ejecutable.

Este comando admite la entrada desde un fichero ó bien desde el teclado. Normalmente, le daremos la entrada usando el "here document" de la shell.

El "cuando" admite formas complejas de tiempo. En vez de contar todas, veamos algunos ejemplos que seguro que alguno nos cuadrará:

* Ejecutar la shell "salu2.sh" que felicita a todos los usuarios, pasado mañana a las 4 de la tarde:

at 4pm + 2 days <<EOF

/usr/yo/salu2.sh

EOF

* Lanzar ahora mismo un listado:

at now + 1 minute <<EOF

"lp -dlaserjet /tmp/balance.txt"

EOF

* Ejecutar una cadena de reindexado de ficheros larguísima y apagar la máquina:

at now + 3 hours <<EOF

"/usr/cadenas/reind.sh 1>/trazas 2>&1; shutdown -h now"

EOF

* A las 10 de la mañana del 28 de Diciembre mandar un saludo a todos los usuarios:

at 10am Dec 28 <<EOF

wall "Detectado un virus en este ordenador"

EOF

* A la una de la tarde mañana hacer una salvita:

at 1pm tomorrow <<EOF

/home/Salvas/diario.sh

EOF

De la misma manera que el comando nohup, estos comandos de ejecución en diferido mandan su salida al mail, por lo que hay que ser cuidadoso y redirigir su salida a ficheros personales de traza en evitación de satu rar el directorio /var/mail.

El comando "at" asigna un nombre a cada trabajo encolado, el cual lo podemos usar con opciones para consultar y borrar trabajos:

at -l: lista todos los trabajos en cola, hora y día de lanzamiento de los mismos y usuario.

at -d <trabajo>: borra <trabajo> de la cola.

Ya que el uso indiscriminado de este comando es potencialmente peligroso, existen dos ficheros que son inspeccionados por el mismo para limitarlo: at.allow y at.deny. (Normalmente residen en /usr/spool/atjobs ó en /var/a t).

at.allow:si existe, solo los usuarios que esten aquí pueden ejecutar el comando "at".

at.deny:si existe, los usuarios que estén aquí no estan autorizados a ejecutar "at".

El "truco" que hay si se desea que todos puedan usar at sin tener que escribirlos en el fichero, es borrar at.allow y dejar sólo at.deny pero vacío.

- Batch.

Ejecuta el comando como en "at", pero no se le asigna hora; batch lo ejecutará cuando detecte que el sistema se halla lo suficientemente libre de tareas. En caso que no sea así, se esperará hasta que oc urra tal cosa.

-Cron.

No es un comando; es un "demonio", ó proceso que se arranca al encender la máquina y está permanentemente activo. Su misión es inspeccionar cada minuto los ficheros crontabs de los usuarios y ejec utar los comandos que allí se digan a los intervalos horarios que hayamos marcado. Como se acaba se señalar, los crontabs son dependientes de cada usuario.

Se deben seguir los siguientes pasos para modificar, añadir ó borrar un fichero crontab:

1 - Sacarlo a fichero usando el comando "crontab -l >/tmp/mio", por ejemplo.

2 - Modificarlo con un editor de acuerdo a las instrucciones de formato que se explican a continuación.

3 - Registrar el nuevo fichero mediante el comando "crontab /tmp/mio", por ejemplo.

mira cada minuto el directorio /var/spool/crontabs y ejecuta los comandos

crontab.

El formato del fichero es el siguiente:

#minutos horas dia mes mes dia-semana comando (# = comentario)
minutos : de 0 a 59.
horas : de 0 a 23.
dia del mes : de 0 a 31.
mes : de 0 a 12.
día semana : de 0 a 6 (0 = domingo, 1 = lunes...)

Aparte de la especificación normal, pueden utilizarse listas, es decir, rangos de valores de acuerdo con las siguientes reglas:

8-11 : Rango desde las 8 hasta las 11 ambas inclusive.
8,9,10,11 : Igual que antes; desde las 8 hasta las 11.

Si queremos incrementar en saltos diferentes de la unidad, podemos escribir la barra "/":

0-8/2 : Desde las 12 hasta las 8 cada 2 horas. Equivale a 0,2,4,6,8.

El asterisco significa "todas" ó "todo". Por tanto:

*/2 : Cada dos horas.

Ejemplos más evidentes:

ejecutar una salvita todos los dias 5 minutos despues de las 12 de la noche:

5 0 * * * /home/salvas/diaria.sh 1>>/home/salvas/log 2>&1
ejecutar a las 2:15pm el primer dia de cada mes:
15 14 1 * * /home/salvas/mensual.sh 1>/dev/null 2>&1
ejecutar de lunes a viernes a las diez de la noche - apagar la maquina que es muy tarde
0 22 * * 1-5 shutdown -h now 1>/dev/null 2>&1
ejecutar algo cada minuto
* * * * * /home/cosas/espia.sh

4 TRATAMIENTO DE FICHEROS.

Comando :wc <opciones> <nombre-archivo>

Misión: cuenta líneas,palabras o caracteres de un fichero.

opciones:

-l:líneas

-c:letras

-w:palabras

Ejemplo:

l=`wc -l /etc/passwd`; echo "Hay $l usuarios registrados en este equipo"

En la variable $l tendríamos el número de líneas del fichero /etc/passwd.

Comando: tr <opciones> 'caracteres orig' 'caracteres destino' < nombre-archivo

Misión: sustituye globalmente un caracter por otro.

Ejemplo:

tr '[a-z]' '[A-Z]' < /etc/passwd >/tmp/passwd.MAYUSCULAS

Pasa el fichero de entrada a mayúsculas.

Comando: cut <opciones> <nombre de archivo>

Misión: corta texto a través del número absoluto de columna o el número relativo de

campo.

Opciones:

-c:numero absoluto de columna

-f:numero de campo.

-d:delimitador de campo.

Especificación de números de columna.

1,3,10 : primera, tercera y decima columnas.

1-3 : de la primera a la tercera columna.

1-3,8 : de la primera a la tercera y la octava.

-5 : de la primera a la quinta columna

5- : de la quinta a la última.

Ejemplo:

cut -d"." -f1 /etc/passwd

lista sólamente los nombres del fichero /etc/passwd, cortando por el delimitador ':'.

Comando: paste.

Misión:

Comando: split <-numero de líneas de los ficheros> <fichero>

Misión: divide <fichero> en n ficheros de <num.lin> cada uno. Si no se indica el

numero de líneas, se consideran 1000.

los ficheros que genera se llaman xaa, xab, xac...

Ejemplo:

$ split gordo

$ ls

gordo xaa xab xac xad xae xaf xag

Ha partido el fichero "gordo" en ficheros más pequeños, de 1000 líneas cada uno, denominados "xaa","xab"..... Si borramos el fichero "gordo", podríamos volver a ge nerarlo con el comando siguiente:

$ cat xaa xab xac xae xaf xag > gordo

Comando: sort <opciones> <indicador de campo> <fichero>

Misión: ordena un fichero de acuerdo a una secuencia de odenacion determinada.

Ejemplo

dado un fichero con:

codigo nombre ciudad telefono

$ sort fichero # ordena por todos los campos

$ sort +1 fichero # ordena a partir del campo 1 (nombre; codigo es el 0)

$ sort +2n fichero# ordena por nombre, en secuencia invertida.

$ sort +2n -t":" fichero # igual que antes, pero suponiendo que el separador es ":".

Comando: grep <cadena a buscar> <fichero>

Misión: buscar apariciones de palabras ó textos en ficheros.

Ejemplo:

$ grep root /etc/passwd # busca las líneas que contienen "root" en el fichero.

root:Ajiou42s:0:1:/:/bin/sh

Otro:

$ ps -e | grep -v constty # busca qué procesos no ruedan en la consola

(......)

La familia de grep, así como otros comandos tales como de (editor de líneas), sed (editor batch) y vi, manejan expresiones regulares para la búsqueda de secuencias de caracteres. Una expresión regular es un a expresión que especifica una secuencia de caracteres. Un ejemplo de esto es el comando: ls -l | grep "^d"; el "^d" es una expresión regular que equivale a decir "si la letra -d- está al principio de la línea"; la construcción anterior nos lista sólo los directorios.

Las expresiones regulares conviene esconderlas de la shell encerrándolas entre comillas.

Algunos montajes de expresiones regulares son:

<.> (Punto) :cualquier carácter distinto al del fin de línea.

[abc] :la letra a, la b ó la c.

[^abc] :cualquier letra distinta a a, b ó c.

[a-z] :cualquier letra de la -a- a la -z-.

Una letra seguida de un asterisco -*- equivale a cero ó más apariciones de la letra. Por tanto, la expresión.* equivale a decir "cualquier cosa". Por tanto, el comando:

ls -l | grep '\.sh.*"

lista todos los ficheros que terminen en.sh mas los terminados en.sh<otras letras>. Nótese que el punto inicial lo hemos desprovisto de su significado "escapándolo" con un backslash -\-.

^ : al principio de una expresión regular equivale a "desde el principio de la línea".

$: al final de una expresión regular equivale a "hasta el final de la línea".

Ejemplos de esto:

ls -l | grep '\.sh$' : saca los ficheros que SOLO acaben en.sh

Este tema se complica todo lo que se quiera. Veamos algunas expresiones regulares bastante usadas en comandos tipo sed (que veremos más tarde) para percatarse de ello:

[^] : cualquier letra diferente de espacio.
[^]* : una palabra cualquiera (varias letras no blancas).
[^]* * : una palabra seguida de un número incierto de blancos.
^[^]* * : la primera palabra de la línea y a todos los blancos que la siguen.
Comando: diff <fichero1> <fichero2>
Misión: Encuentra las diferencias entre dos ficheros.
Ejemplo:
$ cat nombres1
jose
juan
roberto
$ cat nombres2
jose
roberto
$ diff nombres1 nombres2

2d1
< juan
Comando: join <fichero1> <fichero2>
Misión: mezcla ficheros clasificados
$ cat uno
antonio alperchines,5
juan seneca,32
maria oxigeno,45
pablo isaac peral,54
$ cat dos
antonio madrid
juan segovia
maria avila
pablo guadalajara
$ join uno dos
antonio alperchines,5 madrid
juan seneca,32 segovia
maria oxigeno,45 avila
pablo isaac peral,54 guadalajara

Comando: fold <-ancho-en-columnas> <fichero>

Misión: pasar palabras automáticamente a la línea siguiente. En el caso de tener un texto con líneas de más de 80 caracteres, al imprimirlo queda truncado. Usando este comando, cualquie r texto que sobrepase el ancho establecido se pasa a la línea inferior.

Ejemplo:

fold -80 carta.txt | lp

Comando: sed 'comandos de sed' <ficheros>

Misión efectúa cambios en 'fichero', enviando el resultado por la salida estándar. Pueden guardarse los resultados de estas operaciones en un fichero como siempre, redireccionando la salida, tal que - sed 'expresion' fichero >/tmp/salida -.

Utilizaremos sed normalmente cuando sea necesario hacer cambios en ficheros de texto; por ejemplo, nos puede valer para cambiar determinada variable en una serie de ficheros de profile de usuario, etc.

La explicación completa de este comando exigiría mucho detalle; veamos sólamente algunos ejemplos de operaciones con sed:

* Cambiar, en el fichero /home/pepito/.profile, la expresión TERM=vt100 por TERM=vt220. Usamos el comando 's/expresion_a_buscar/expresion_a_cambiar/g' (s=search,g=global):

sed 's/TERM=vt100/TERM=vt220/g' /home/pepito/.profile >/tmp/j && mv /tmp/j /home/pepito/.profile

(si el comando ha sido OK, entonces movemos el /tmp/j).

* Meter tres blancos delante de todas las líneas del fichero /tmp/script.sh:

sed 's/^/ /' /tmp/script.sh >/tmp/j && mv /tmp/j /tmp/script.sh

(el ^ identifica el principio de cada línea).

* (mas raro): Cepillarse todos los ficheros de un directorio:

ls | sed 's/^/rm -f /' | sh

(del ls sale el fichero; el sed le pone delante "rm -f" y el sh lo ejecuta.

APENDICE A: Ejemplos de shell-script multipropósito.

```
# programa para contar
# llamada: contar.sh <numero>
case $# in
0)    echo "Falta argumento"
      exit 1
      ;;
esac
c=0
while [ $c -le $1 ]
do
echo $c
c=‘expr $c "+" 1‘
done


# programa para listar sólo subdirectorios
# llamar como "dir.sh" <argumentos opcionales>
ls -l $* | grep ‘^d’
# programa para ver quien esta conectado
# llamar como "ju <usuario> <usuario>..."
for j in $*
do
      donde=`who | grep $j | sed ‘s/^[^ ]* *\([^ ]*\).*/\1/p’`
      if [ "$donde" ]
      then
            echo "$j conectado en $donde"
      else
            echo "$j no esta conectado"
      fi
done
# programa para paginar un fichero con mensajito al final de cada pantalla
# llamar como "mas <ficheros>"
pg -p ‘Pagina %d: ‘ -n $*
# programa para seleccionar interactivamente ficheros
# se usa dentro de una pipe como: cat `pick *` | <el comando que se quiera, tal que
lp>
for j #  no  tiene  error!  esta  cosa  rara  permite  coger  todos  los  ficheros  del
argumento
do
      echo "$j ? \c" >/dev/tty
      read resp
      case $resp in
            s*) echo $j;;     # si escribe "s" ó "si"
            n*) break;; # lo mismo
      esac
done </dev/tty
# programa para borrar de un directorio los ficheros mas viejos de 7 días
# llamar como "limpia.sh <directorio>"
case $# in
0)    echo "Falta argumentos"
      exit 1
      ;;
esac
cd $1 && { find. -type f -mtime +7 -exec rm -f -- {} \;; }
# programa para si los ficheros de un directorio ocupan mas de 5 Mb se truncan.
```

```
# llamar como "trunca.sh <directorio>"
case $# in
0)    echo "Falta argumento"
      exit 1
      ;;
esac
cd $1 || echo "No puedo cambiar a $1 - saliendo"; exit 1
for j in *
do
      if [ ! -f $j ]
      then
            continue
      fi
      siz=`ls -ld $j | awk ‘{ print $5 }’
      if [ $siz -gt 5000000 ]
      then
            echo "Truncando $j"
            >$j
      fi
done
```

programa que saca el date en formato dia/mes/año hora:minuto:segundo

llamar como "fecha.sh"

d=`date ‘+%d/%m/%y %H:%M:%S’`

echo "FECHA Y HORA: $d"

```
# programa para detectar si alguien inicia sesión
# llamar como "watchfor <nombre_de_usuario>"
# Extraído del "Unix Programming Environment" de Kernighan & Pike
case $# in
0) echo "Sintaxis: watchfor usuario";
exit 1;;
esac
until who | grep "$1"
do
sleep 60
done
```

APENDICE B: Optimización y mantenimiento del sistema.

Si bien las tareas específicas de optimización y mantenimiento suelen estar muy "pegadas" a la versión y al dialecto del UNIX, se exponen a continuación unas directrices básicas a modo de or ientación; para desgracia del lector, cualquier detalle adicional habrá de buscarlo en el denostado "System Administrator Guide" de su equipo. Todo lo antedicho puede resumirse en el siguiente aserto: "estos son unos cuanto s consejos caseros".

Lo que aquí se detalla se basa fundamentalmente en System V, algo de BSD 4.4 y en el Fast File System con las mejoras de Berkeley.

Para poder optimizar un sistema operativo ó un conjunto de aplicaciones rodando en él, es necesario conocer con exactitud qué recursos están siendo consumidos, las limitaciones de los mismos y la mejor mane ra de asignarlos a los usuarios y a las aplicaciones. Excesivas demandas, ó nuevas demandas a un sistema ya muy sobrecargado puede afectar significativamente al rendimiento.

En la mayoría de los casos, la solución a estas sobrecargas pasa por alguno de estos puntos:

- Modificar algún parámetro del núcleo (kernel). En algunos casos, existen límites impuestos por el sistema que pueden ser aumentados (número de ficheros abiertos, número de semáforos....) y en otros corresponden a tamaños de búferes, tablas del sistema, etc.

- Redistribuir los datos en los discos.

- Eliminar la fragmentación reconstruyendo los sistemas de ficheros.

- Rediseñar aplicaciones que distribuyan los recursos de la manera más optimizada posible.

De cualquier manera, puede ser necesario añadir más potencia de CPU, más memoria ó más discos. Antes de acometer cualquiera de estas tareas, conviene repasar distintos puntos en busca siempre de conseguir que el equipo ruede de la manera más descargada posible.

* Ficheros de log y similares.

Muchos de los programas dentro de un sistema UNIX guardan información en ficheros de log. Es necesario por tanto hacer limpieza y reducciones periódicamente, puesto que un fichero, cuanto más grande sea, más trabajo cuesta escribir en él.

Examinar antes de nada el fichero "/etc/syslog.conf", que contiene los nombres de aquellos ficheros que intervienen.

Vigilar especialmente los siguientes ficheros y directorios:

/var/adm/log/.....	Ficheros diversos de log del sistema (kernel, tcp/ip...)
/var/adm/sulog	Cada vez que se ejecuta el comando ‘su’, añade un registro aquí.
/var/adm/messages	Recoge los mensajes de error del syslogd (demonio de log).
/etc/log/....	Logs de arranque.
/etc/wtmp,	
/etc/wtmpx	Cada usuario que entra al sistema crea un registro.
/var/cron/log	Arranques del cron.
/usr/preserve	Restos de ficheros editados con el editor ‘vi’
/usr/mail	Ficheros de correo de todos los usuarios.

Consultar el "System Administration Guide" para detalles específicos.

* Shell-scripts.

-En muchos casos, el acceso al sistema implica ejecutar una aplicación. Si se arranca la shell de usuario y después la aplicación, tendremos DOS procesos iniciales por cada usuario. Podemos ver que esto es así usando el comando "whodo". Si este es el caso, deberemos modificar el.profile del usuario de manera que este ejecute la aplicación con el comando ‘exec <nombre de aplicación>‘, de tal manera que el proceso de aplicación reemplaza el proceso de la shell.

-Las "shell-scripts" son muy fáciles de usar y crear, pero gastan más recursos que un programa compilado. Si su uso es intensivo en las aplicaciones, es conveniente estudiar cuales se pueden pasar a ejecutables C.

-La variable "PATH", usada para localizar ejecutables y shells, debe contener sólo aquellas entradas que realmente sean necesarias y los directorios más usados han de ser los que aparezcan primero.

* Procesos / Sistemas de ficheros / discos duros.

- En ningún caso permitir que un sistema de ficheros llegue a ocupar más del 90 %.

- Repartir aquellos directorios muy grandes en varios más pequeños; a ser posible, evitar aquellos directorios con más de 320 entradas; podemos buscarlos con el siguiente comando:

```
find / -type d -size +10 -print
```

- Buscar procesos que "coman" mucho tiempo de procesador; pueden ser programas buclados ó simplemente programas que ejecutan intensivamente procesos algebraicos en coma flotante ó con mucha recursividad; el pro grama benchmark de Dhrystones por si solo ocupa en una máquina corriente UNIX el 95 % de CPU.

Podemos averiguar qué procesos gastan más mediante la siguiente operación:

ps -ef >/tmp/ps.1;sleep 20;ps -ef >/tmp/ps.2;diff /tmp/ps.1 /tmp/ps.2

(El comando "diff" saca aquellos procesos que han consumido tiempo de CPU).

- Lanzar procesos relacionados con cintas fuera de horas de trabajo (Son pesadísimos para el sistema).

- Aquellos sistemas de ficheros susceptibles de fragmentarse en exceso (uso pesado haciendo borrados, adiciones y modificaciones de registros), se pueden desfragmentar copiándolos a cinta, borrando todo el sistema de fichero s y volviéndolos a recuperar, tarea esta que conviene aplicar periódicamente.

- Si bien UNIX da siempre mayor prioridad a los procesos "on-line", aquellas cadenas batch que sean gravosas para el sistema (listados gordos, procesos de reconstrucción...) deberán ser ejecutadas con menor p rioridad, usando a tal efecto el comando "nice" ó "renice" para porcesos ya en curso.

- A medida que aumentamos discos duros, conviene aumentar también de controladoras. Mientras que un disco puede procesar de 25 a 35 operaciones de E/S por segundo, una controladora normal procesa unas 65; si bien una placa de estas aguanta hasta 7 discos, es conveniente no pasarse.

- Colocar los sistemas de ficheros más usados sobre los discos más rapidos, en concreto el /tmp y el /var, así como las áreas de swap;conviene que estén repartidas sobre varios discos.

- Colocar la informacion mass utilizada en zonas adyacentes.

CONCLUSION. Optimizar un sistema UNIX no es ni fácil ni evidente; normalmente, suele bastar con mantenerlo saneado a base de operaciones periódicas tales como: - Apagar el equipo al menos semanalmente. - Mantener los directorios /tmp y /var/tmp descargados. - Borrar y truncar aquellos ficheros de log que crecen (ver sección anterior), bien al arrancar el equipo ó bien "sobre la marcha", poniendo una entrada en el cron. - Buscar y eliminar ficheros supérfluos ó innecesarios, sacando a cinta aquellos que no vamos a utilizar a menudo.

Vuelta a la Pagina Principal _