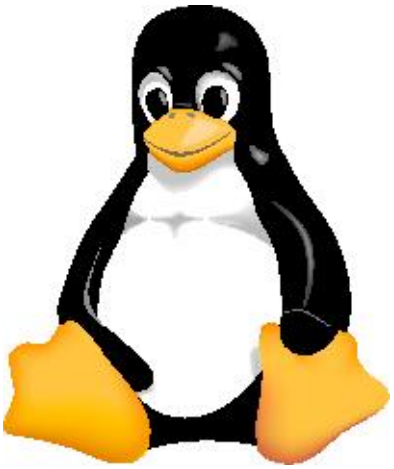

Comandos linux unix y programación shell

4 PARTY

Documento de la charla de Iñigo Tejedor & Pello Altadill
http://4party.cuatrovientos.org

Índice de contenido

1.INTRODUCCIÓN....3
El mapa de ficheros y directorios en linux....4
2.El shell....5
Buscar ayuda....5
3.Guía de comandos....6
Comandos básicos y manejo de ficheros....6
Comandos de administración: usuarios, procesos, kernel....9
Comunicaciones....10
Comunicación entre procesos....12
Redirección de entrada/salida....12
Compresión de ficheros y directorios....13
Accediendo a dispositivos y particiones....14
4.Usando el shell....16
Las flechas....16
El historial de comandos....16
Relleno de comandos y ficheros....17
Comodines....18
Ejecución de comandos....18
Comandos en segundo plano....19
5.Programación de scripts de shell....22

Introducción 22
Variables.....23
arrays.....24
Otros usos.....24
Operaciones.....25
Aritméticas.....25
Lógicas.....26
Condicionales.....27
case.....30
Iteraciones.....31
for.....31
while.....32
until.....33
select.....34
Funciones.....34
Librerías.....35
Señales.....36
Colores.....36
6.Bibliografía, referencias y enlaces.....38

1.INTRODUCCIÓN

Este guía no es para leer de forma seguida, este guía pretende ser una referencia de comandos Unix/Linux y de programación de scripts de shell que se pueda consultar en cualquier momento. No pretende ser una guía exhaustiva, sino una referencia que sirva como recordatorio de los comandos más utilizados.

¿Y qué es Linux?
Los sistemas operativos del tipo Linux esta formado por procesos y ficheros. Y no hay nada más. Los dispositivos como el disco, el cdrom, la pantalla, esta representado como un fichero en el sistema linux, dentro de /dev. Los sockets de comunicación son ficheros. Los directorios son ficheros. Los ficheros son ficheros.

El mapa de ficheros y directorios en linux

Hay directorios que pueden cambiar segun la distribucion que sea. Las de linux se parecen bastante, pero en general tienen lo mismo.

/ +-**/bin** : los binarios MAS basicos como: ls, cd, pwd, cp, rm
+-**/boot** : archivos de arranque, imagen de kernel.
+-**/cdrom** : punto de montaje del cdrom
+-**/etc** : configuracion. Todas las configuraciones (red,scripts de inicio,firewall,usuarios)
+-----**/mail/** : configuracion de sendmail
+-----**/rc.d/** : ficheros con scripts de inicio de sistema
+-----**/sysconfig/** : configuracion de sistema (red,rutas,interfaces)
+-**/dev** : todos los dispositivos de sistema (pantalla,raton,impresora,disco duro) representados por un fichero
+-**/home** : directorios particulares de los usuarios
+-**/lib** : librerias de sistema, modulos de kernel
+**-/lost+found** : directorio en el que se guarda contenido perdido tras un chequeo de disco
+-**/mnt** : punto de montaje (opcional) para particiones locales, remotas (nfs),..
+-**/proc** : sistema de ficheros que refleja el estado y configuracion del sistema
+-**/sbin** : binarios basicos que influyen en configuraciones (firewall, rutas,) de kernel
+-**/usr** : binarios de servidores, programas, manuales, documentos, etc (este ocupa el que mas)
+-**/opt** : binarios opcionales o programas que no se instalan desde el principio.
+-**/root** : directorio home de superusuario.
+-**/tmp** : directorio temporal, utilizado para guardar sesiones, sockets, etc..
+-**/var** : directorio en el que se guarda informacion variable: logs, BBDD (postgres, mysql)
+-----**/log/** : todos los logs de sistema y servidores
+-----**/lib/pgsql/** : postgres
+-----**/lib/mysql/** : mysql
+-----**/spool/mqueue/** : cola de correo del servidor (entregas sendmail)
+-----**/spool/mail/** : buzones de correo

2.El shell

Al entrar en un Linux en modo consola estamos usando un shell o interprete de comandos. Un shell no es más que un programa que le facilita al usuario interactuar con el sistema operativo: administrar el sistema, programar, crear todo tipo de contenidos de texto, etc... para hacer todo eso el usuario debe utilizar una serie de comandos que le permiten manipular ficheros, procesos, etc...

En los sistemas Windows el shell es un entorno visual intuitivo y fácil de utilizar, aunque también tiene otro shell disponible, una herencia del sistema MS-DOS que no ha evolucionado mucho. En Linux disponemos de multitud de entornos visuales para elegir, pero la gran diferencia respecto a Windows es que en Linux el entorno gráfico no es más que un programa más; es algo opcional que podemos lanzar desde un shell o intérprete de comandos. La otra diferencia importante es que el shell de linux es una herramienta muy poderosa.

Existen distintos interpretes de comandos en el mundo Unix: csh, bash, tsh, ksh,.. pero salvo pequeñas diferencias todos son parecidos. En este documento partimos sobretodo de bash2, ya que esta muy extendido a través de Linux.

Buscar ayuda

Todos los comandos tienen ayuda sobre sus opciones y más de una forma de llegar a esa ayuda:

man comando : a través de man accedemos a la página de manual de cualquier programa. El manual de comandos nos da TODA la información de un comando e incluso al final nos sugiere otros comandos similares o relacionados. Es la forma clásica de acceder a la ayuda extendida aunque para los usos más frecuentes de comandos siempre se puede usar opciones más simples que pueden variar de un comando a otro:

comando -h
comando –help
comando -?
info comand

3.Guía de comandos

Comandos básicos y manejo de ficheros

cd
Descripción: =chage dir. comando para cambiarnos de directorio.
Ejemplos: cd, cd /ruta/de/directorio, cd ../../directorio/

pwd
Descripción: nos dice en qué directorio nos encontramos actualmente
Ejemplos: pwd

ls
Descripción: =list. listar contenido de directorios.
Ejemplos: ls, ls -l, ls -fl, ls --color

cp
Descripción: =copy. copiar ficheros/directorios.
Ejemplos:cp -rfp directorio /tmp, cp archivo archivo_nuevo

rm
Descripción: =remove. borrar ficheros/directorios.
Ejemplos: rm -f fichero, rm -rf directorio, rm -i fichero

mkdir
Descripción: =make dir. crear directorios.
Ejemplos: mkdir directorio

rmdir
Descripción: =remove dir. borrar directorios, deben estar vacios.
Ejemplos: rmdir directorio

mv
Descripción: =move. renombrar o mover ficheros/directorios.
Ejemplos: mv directorio directorio, mv fichero nuevo_nombre, mv fichero a_directorio

nano
Descripción: editor de fichero muy simple. Vi o emacs son para la 31337.
Ejemplo: nano -w

clear
Descripción: limpia la pantalla. Ctrl-L tiene el mismo efecto.

exit
Descripción: termina la sesión del shell. Ctrl-D tiene el mismo efecto.

date
Descripción: gestión de fecha de sistema, se puede ver y establecer.
Ejemplos: date, date 10091923

history
Descripción: muestra el historial de comandos introducidos por el usuario.
Ejemplos: history | more

more
Descripción: muestra el contenido de un fichero con pausas cada 25 lineas.
Ejemplos: more fichero

Al hacer **more**:
/cadena : podemos hacer búsqueda de cadena
f : adelante
b: volver arriba
v: iniciar vi en la linea que estamos

Nota: estas opciones también sirven para el comando **man**

grep
Descripción: filtra los contenidos de un fichero.
Ejemplos:cat fichero | grep cadena, grep -il “cadena” directorio/

cat
Descripción: muestra todo el contenido de un fichero sin pausa alguna.
Ejemplos: cat fichero

chmod
Descripción: cambia los permisos de lectura/escritura/ejecucion de ficheros/directorios.
Ejemplos: chmod +r fichero, chmod +w directorio, chmod +rw directorio -R, chmod -r fichero
chmod +x fichero : da permiso de ejecucion

chown
Descripción: =change owner. cambia los permisos de usuario:grupo de ficheros/directorios.
Ejemplos: chown root:root fichero, chown pello:usuarios directorio -R

tar
Descripción: =Tape ARchiver. archivador de ficheros.
Ejemplos: tar cvf fichero.tar directorio , tar xvf fichero.tar, tar zcvf fichero.tgz directorio, tar zxvf fichero.tgz

gunzip
Descripción: descompresor compatible con ZIP.
Ejemplos: gunzip fichero

rpm
Descripción: gestor de paquetes de redhat y fedora. Para instalar o actualizar software de sistema.
Ejemplos: rpm -i paquete.rpm, rpm -qa programa, rpm --force paquete.rpm, rpm -q --info programa

dpkg
Descripción: gestor de paquetes de Debian. Para instalar o actualizar software en el sistema.
Ejemplos: dpkg -i paquete.deb

alias
Descripción: para crear alias de comandos. Útil para comandos largos.

mount
Descripción: montar unidades de disco duro, diskette, cdrom.
Ejemplos: mount /dev/hda2 /mnt/lnx, mount /dev/hdb1 /mnt -t vfat

umount
Descripción: desmontar unidades.
Ejemplos: umount /dev/hda2, umount /mnt/lnx

who
Descripción: muestra los usuarios de sistema que han iniciado una sesion.
Ejemplos: who, w, who am i

sort
Descripción: ordena el contenido de un fichero.
Ejemplos: cat /etc/numeros | sort, ls | sort

ln
Descripción: =link. para crear enlaces, accesos directos.
Ejemplos: ln -s /directorio enlace

tail
Descripción: muestra el final (10 lineas) de un fichero.
Ejemplos:tail -f /var/log/maillog, tail -100 /var/log/maillog | more

head
Descripción: muestra la cabecera (10 lineas) de un fichero.
Ejemplos: head fichero, head -100 /var/log/maillog | more

file
Descripción: nos dice de que tipo es un fichero.
Ejemplos: file fichero, file *

cmp
Descripción: compara dos ficheros y nos dice si son distintos
Ejemplos: cmp fichero1 fichero2

file
Descripción: nos dice de que tipo es un fichero.
Ejemplos: file fichero, file *

diff
Descripción: muestra las diferencias entre dos ficheros. Muy usado para parchear software.
Ejemplos: diff fichero1 fichero2

wc
Descripción: word count, calcula número de palabras y otros datos similares de un fichero.
Ejemplos: wc fichero, wc -l fichero

Comandos de administración: usuarios, procesos, kernel

sysctl
Descripción: Configurar los parámetros del kernel en tiempo de ejuecución.
Ejemplos: sysctl -a

ulimit
Descripción: muestra los limites del sistema (maximo de ficheros abiertos, etc..)
Ejemplos: ulimit

adduser
Descripción: añadir usuario de sistema.
Ejemplos: adduser pepe, adduser -s /bin/false pepe

userdel
Descripción: = eliminar usuario de sistema
Ejemplos: userdel pepe

usermod
Descripción: = modificar usuario de sistema
Ejemplos: usermod -s /bin/bash pepe

df
Descripción: = disk free. espacio en disco disponible. Muy util.
Ejemplos: df, df -h

uname
Descripción: =unix name. Informacion sobre el tipo de unix en el que estamos, kernel, etc.
Ejemplos: uname, uname -a

netstat

Descripción: la informacion sobre las conexiones de red activas.
Ejemplos: netstat, netstat -ln, netstat -l, netstat -a

ps
Descripción: =proccess toda la informacion sobre procesos en ejecucion.
Ejemplos: ps, ps -axf, ps -A, ps -auxf

pstree
Descripción: =proccess tree, muestra los procesos en forma de árbol
Ejemplos: pstree

kill
Descripción: envía señales a procesos. La más común es la de matar el proceso.
Ejemplo: kill -9 34 (la señal -9 es KILL y mata el proceso numero 34)

free
Descripción: muestra el estado de la memoria RAM y el SWAP.
Ejemplos: free

vmstat
Descripción: muestra el estado de la memoria virtual
Ejemplos: vmstat, vmstat -s

du
Descripción: =disk use. uso de disco. Muestra el espacio que esta ocupado en disco.
Ejemplos: du *, du -sH /*, du -sH /etc

lsof
Descripción: muestra los ficheros(librerias, conexiones) que utiliza cada proceso
Ejemplos: lsof, lsof -i, lsof | grep fichero

lsmod
Descripción: Muestra los modulos de kernel que estan cargados.
Ejemplos: lsmod

insmod
Descripción: instala modulos de kernel
Ejemplo: insmod e1000, insmod usb_core

modprobe
Descripción: Trata de instalar un modulo, si lo encuentra lo instala pero de forma temporal.
Ejemplos: modprobe ip_tables, modprobe eepro100

rmmod
Descripción: Elimina modulos del kernel que estan cargados
Ejemplos: rmmod <nombre de modulo>

fdisk

Descripción: sirve para gestionar las particiones de una unidad de disco
Ejemplos: fdisk /dev/hda , fdisk -l /dev/sda

Comunicaciones

telnet
Descripción: Establece conexiones a puertos TCP
Ejemplo: telnet localhost 25

mesg
Descripción: Establece si se aceptan mensajes a través de write o talk.
Ejemplo: mesg -y

write
Descripción: envía mensajes a otros usuarios.
Ejemplo: write jrmorris pts/0

wall
Descripción: envía un mensaje a todos los usuarios conectados al sistema
Ejemplo: wall “a cascarla el sistema”

ifconfig
Descripción: =interface config. configuracion de interfaces de red, modems, etc.
Ejemplos: ifconfig, ifconfig eth0 ip netmask 255.255.255.0

route
Descripción: gestiona las rutas a otras redes.
Ejemplos: route, route -n

iptraf
Descripción: muestra en una aplicacion de consola TODO el trafico de red IP, UDP, ICMP.
Permite utilizar filtros, y es SUMAMENTE UTIL para diagnostico y depuracion de firewalls
Ejemplos: iptraf

tcpdump
Descripción: vuelca el contenido del trafico de red.
Ejemplos: tcpdump, tcpdump -u

ping
Descripción: heramienta de red para comprobar entre otras cosas si llegamos a un host remoto.
Ejemplos: ping www.rediris.es

traceroute
Descripción: herramienta de red que nos muestra el camino que se necesita para llegar a otra maquina.
Ejemplos: traceroute www.rediris.es

mtr
Descripción: Matt's Trace Route, un traceroute de consola algo más visual, muy util para observar donde hay pérdidas de paquetes.

mail
Descripción: envio y lectura de correo electronico.
Ejemplos: mail pepe@cuatrovientos.org < fichero, mail -v pepe@cuatrovientos.org < fichero

wget
Descripción: programa para descargar ficheros por http o ftp.
Ejemplos: wget http://www.rediris.es/documento.pdf

lynx
Descripción: navegador web con opciones de ftp, https.
Ejemplos: lynx www.cuatrovientos.org, lynx --source http://www.cuatrovientos.org/script.sh | sh

ftp
Descripción: cliente FTP.
Ejemplos: ftp ftp.cuatrovientos.org

whois
Descripción: whois de dominios.
Ejemplos: whois cuatrovientos.org

sniffit
Descripción: Sniffer o husmeador de todo el trafico de red. No suele venir instalado por defecto.
Ejemplos: sniffit -i

Comunicación entre procesos

Tuberias o pipes: | sirven para comunicar procesos, usualmente para pasar el resultado **de un proceso a otro proceso**: por ejemplo ps axf | grep bash: lista todos los procesos y filtra los que se llamen bash . La tubería en definitiva lo que hace es unir la salida de un comando con la entrada de otro.

Redirección de entrada/salida

`proceso > fichero` : con este símbolo podemos redirigir la salida estándar de un comando a un fichero. Téngase en cuenta una cosa. Si decimos fichero siempre lo vamos a decir de manera genérica, puede ser un fichero de texto o la pantalla de terminal, ahí cabe TODO.

```
linux~$ ps -axf > procesos.txt
linux~$ more procesos.txt
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:00 init [2]
    2 ?        SN     0:00 [ksoftirqd/0]
    3 ?        S<     0:01 [events/0]
    4 ?        S<     0:00  \_ [khelper]
...
```

Nota: para evitar accidentes con la redirección `>`, puede establecerse una opción en el shell que se llama noclobber.

`procesa |> fichero` : con esto redirigimos el resultado a un fichero, sobrescribiéndolo incluso con la opción de shell noclobber activada.

`procesa >> fichero` : con esto redirigimos el resultado a un fichero, pero sin sobrescribirlo, lo que hacemos es escribir al final de este (append en ingles).

`procesa < fichero` : con esto redirigimos el contenido del fichero a un programa. Se usa para utilizar el contenido de un fichero como input de un comando.

```
linux~$ procesa < fichero.txt
```

`proceso << END` : este redirección se utiliza para iniciar el paso de parámetros a un programa, y se termina cuando escribimos "END" o cualquier otra palabra que hayamos especificado al inicio del comando. Un ejemplo clásico sería una sesión FTP automatizada a la que le direccionamos todos los comandos de golpe. Al aparecer de nuevo la palabra FINAL se terminan de aceptar datos y se ejecuta el comando.

```
linux~$ ftp -inv << FINAL
open ftp.rediris.es
user usuario password
cd /debian
ls
```

```
exit
FINAL
...
```

`proceso <> fichero` : esta redirección permite que un proceso utilice un fichero tanto para leer como para escribir:

```
linux~$ proceso <> fichero
```

Compresión de ficheros y directorios

Existen varias opciones de compresión de ficheros en Linux: gzip, bz2, rar, zip, y todos
ellos se pueden combinar con el empaquetar o archivador TAR.

tar

Archivador, agrupa ficheros en uno, además se le puede decir que comprima

tar cfp resultado.tar /etc /var : guarda el contenido de /etc y /var dentro del fichero resultado.tar
tar xfp resultado.tar : saca todo el contenido de resultado.tar
tar zcfp usr.tgz /usr : archiva y comprime con gzip el directorio /usr
tar jcfp usr.tgz /usr : archiva y comprime con bzip2 el directorio /usr
tar zxfp usr.tgz : DEScomprime el fichero anterior

gzip/gunzip, zip/unzip, rar/unrar

Compresor/Descompresor. Los ficheros comprimidos con gzip o con zip no se descomprimen igual. Un fichero comprimido con el winzip habría que abrirlo con el unzip en linux.

gzip fichero : comprime fichero, le añade la extensión gz
gunzip fichero.gz : descomprime ficheros gz.

zip -r9 todo.zip directorio/ : comprime el directorio en el fichero todo.zip
unzip : descomprime para ficheros zip

bz2/bunzip2

Compresor cañero, comprime bastante más que gzip, aunque tarda y chupa mas cpu
bz2 fichero : comprime fichero, le añade la extensión bz2
bunzip2 fichero.bz2: descomprime

---

tar jcfp usr.tar.bz2 /usr: archiva y comprime con bz2

Accediendo a dispositivos y particiones

Para acceder a diskettes, cdroms, dvds, pendrives, cualquier dispositivo USB, etc... es necesario “montarlo” de forma manual, salvo que todo esté configurado de una forma cómoda con automount.

En linux hay un único sistema de ficheros, y si se mete un CD o un USB en el sistema hay que montarlo sobre algún directorio del sistema. Existen algunos directorios predefinidos para esto, aunque en principio lo podemos montar donde nos de la gana.

Para montar un diskette por ejemplo:

```
linux~# mount /dev/fd0 /floppy
```

Para montar un cdrom o dvd

```
linux~# mount /dev/cdrom /mnt
linux~# mount /dev/dvd /montaje/dvd

(el contenido del dvd queda dentro de /montaje/dvd)
```

Para montar un USB basta con usar el dispositivo /dev/sda o /dev/sda1. Linux suele emular los dispositivos USB como discos SCSI. Dependiendo de cómo estén formateados hay que montarlos de una forma u otra. Siempre se puede comprobar cómo esta el particionado usando fdisk.

```
linux~# mount /dev/usb /mnt
```

Para acceder a otras particiones o discos del sistema se siguen los mismos pasos, puede que dependiendo del tipo de partición sea necesario especificarlo.

```
linux~# mount /dev/hda3 /mnt -t vfat
linux~# mount /dev/hda1 /montaje/ -t ntfs -o ro
```

Cuando dejamos de usar el dispositivo y en el caso de que se pueda sacar hay que desmontarlo. El comando es similar: umount y basta con especificar o el dispositivo o el directorio donde se ha montado.

```
linux~# umount /dev/hda3
```

4.Usando el shell

Nada más logearnos en un sistema linux nos enfrentaremos o visto de otro modo nos beneficiaremos del shell y de toda su potencia:

```
linux~$
```

Ese el árido aspecto que puede tener el shell: un prompt a la espera de que metamos comandos. antes de salir ese prompt, el bash comprueba si dentro del directorio del usuario existe algún fichero de inicialización como .bashrc o .profile, y en caso de existir los ejecuta: a través de esos ficheros se establecen algunas variables, alias de comandos, aspecto del prompt, etc...

Para empezar podemos echar un ojo a las variables de entorno del sistema:

```
linux~$ set
BASH=/bin/bash
BASH_VERSINFO=([0]="2" [1]="05b" [2]="0" [3]="1" [4]="release"
[5]="i386-pc-linux-gnu")
BASH_VERSION='2.05b.0(1)-release'
COLUMNS=80
DIRSTACK=()
EUID=1000
GROUPS=()
HISTFILE=/home/usuario/.bash_history
HISTFILESIZE=500
...
```

En el shell ya podemos empezar a meter comandos para hacer la tarea que sea. Muchas veces escribir comandos de texto puede resultar una tarea tediosa y repetitiva, por eso el bash dispone de multitud de facilidades para que no tengamos que escribir tanto.

Las flechas

A través de las teclas cursoras podemos volver a los comandos que hemos introducido anteriormente, algo que los usuarios de MSDOS recordarán como el DOSKEY.
En cualquier momento podemos movernos al inicio o al final del comando podemos usar las teclas Inicio o fin, o en su defecto Ctrl-A o Ctrl-E.

El historial de comandos

A través del comando history podemos echar un ojo a los últimos comandos que hemos ejecutado y podemos repetir cualquiera de ellos gracias al comando especial !

```
linux~$ ls
Desktop  php  script
linux~$
linux~$
2  ls
3  mkdir scripts
4  mkdir php
5  mkdir Desktop
6  ls
7  history
linux~$ !6
Desktop  php  script
linux~$
```

Si simplemente recordamos que habiamos ejecutado un comando que empezaba por hi podemos ejecutarlo haciendo:

```
linux~$ !hi
```

Relleno de comandos y ficheros

Escribir comandos y nombres de ficheros enteros puede ser un rollo. En el shell podemos usar el tabulador en cualquier momento para que nos rellene los comandos y nos complete las rutas de ficheros con lo que nos ahorarremos un montón de trabajo. Por ejemplo:

```
linux~$ mo
(tabulamos y nos muestra todas las posibilidades)
moc                 more                mozilla-1.7.8
moc-qt3             mount               mozilla-firefox
mogrify             mountpoint          mozilla-suite
montage             mozilla             mozilla-thunderbird
linux~$ mou
(tabulamos)
linux~$ mount
```

A la hora de completar los nombres de ficheros y rutas de directorios la tabulación resulta enormemente práctica:

```
linux~$ ls /lib/mo
(tabulación)
linux~$ ls /lib/modules/
(tabulación)
linux~$ ls /lib/modules/2.6.8-2-386
```

Gracias a la tabulación nos ahorramos la tediosa tarea de escribir largos textos.

Comodines

El shell reconoce una serie de caracteres comodín para poder referirse a un conjunto de ficheros o directorios

- * : se corresponde con cualquier contenidos
- ? : se corresponde con un único carácter
- [] : se corresponde con un conjunto de caracteres

```
linux~$ ls
programa.c test.c texto.txt ejemplo.pdf test.o
linux~$ ls *.c
programa.c test.c
linux~$ ls *.?
programa.c test.c test.o
```

Ejecución de comandos

Para ejecutar un comando en el shell basta con ponerlo, y si el comando se encuentra dentro de los directorios indicados en la variable PATH se ejecutará sin problemas.

```
linux~# uptime
 20:49:05 up  2:35,  2 users,  load average: 0.00, 0.00, 0.00
linux~#
```

Podemos encadenar más de un comando en la linea separandolos con ;

```
linux~# uptime;uname -a;date
 20:49:37 up  2:35,  2 users,  load average: 0.00, 0.00, 0.00
Linux 4vientos 2.6.8-2-386 #1 Tue Aug 16 12:46:35 UTC 2005 i686
GNU/Linux
```

```
dom jun  3 20:49:37 CEST 2007
linux~#
```

Si todo va bien, el comando hará lo que tenga que hacer e implícitamente devolverá un valor booleano TRUE. Sabiendo eso podemos encadenar comandos de manera condicional usando && y ||

```
linux~# ls fichero && more fichero
```

Cuando usamos && estamos haciendo que el segundo comando se ejecute **solamente** si el primero se ha ejecutado con éxito. Es un and lógico, podemos encadenar más de dos comandos.

```
linux~# ls fichero && cat fichero && echo "sin problemas OK"
```

Si usamos || el siguiente comando se ejecutará **solamente** si el primero no se ha ejecutado correctamente o ha devuelto algún error.

```
linux~# ls fichero || echo "El fichero no existe"
```

Podemos combinar los dos:

```
linux~# ls fichero && cat fichero || echo "No hay fichero"
```

Comandos en segundo plano

Los sistemas linux son mutitarea: podemos ejecutar más de un comando a al vez. Podemos dejar un comando que se ejecute en segundo plano y mientras podemos ejecutar más. Para dejar un comando en segundo plano simplemente metemos un & al final del mismo. Por ejemplo podemos comprimir un directorio y dejar la operación en segundo plano:

```
linux~# tar zcfp etc.tar.gz /etc &
[2] 4376
linux~#
```

En cualquier momento podemos llevar el comando a primer plano mediante el comando `fg`. Asimismo lo podemos pausar con `Ctrl-Z` y reanudarlo en segundo plano con `bg`.

```
linux~# fg
tar zcfp usr.tar.gz /usr
```

```
(Ctrl-Z)
[2]+  Stopped                 tar zcfp usr.tar.gz /usr
linux~# bg
[2]+ tar zcfp usr.tar.gz /usr &
linux~#
```

Si queremos terminar alguno de los procesos que tenemos en marcha debemos mandarle una señal. Para eso se usa el comando kill, que a pesar de su nombre no es que sirva solo para matar procesos, en realidad sirve para enviar señales a los procesos. La señal más frecuente es KILL o 9:

```
linux~# ps
  PID TTY          TIME CMD
 3593 pts/0    00:00:00 bash
 3767 pts/0    00:00:00 xmms
 3849 pts/0    00:00:00 ps
linux~# kill -9 3767
linux~# ps
  PID TTY          TIME CMD
 3593 pts/0    00:00:00 bash
 3849 pts/0    00:00:00 ps
linux~#
```

Para conocer las señales disponibles podemos echar un ojo en el manual de kill con el comando `man kill`.

Comandos built-in del shell

El shell dispone de un conjunto de comandos que vienen de serie. Son comandos muy sencillos que pueden ayudar en la programación de scripts. Vamos a listarlos, pueden usarse dentro de un script o desde el propio prompt:

- `!` : negación
- `#` : comentarios
- `:` : comando nulo
- `.` : el punto sirve para ejecutar ficheros. Con los scripts del mismo directorio se hace: ./script.sh
- `alias`: sirve para crear alias de comandos largos: alias dir='ls -l | more'
- `bind` : gestiona la librería readline
- `bg` : pasa procesos a background o segundo plano
- `break` : fuerza la salida de fors, whiles, selects o untils

- `builtin`: permite modificar el comportamiento de comandos
- `cd` : el cambio de directorio
- `command` : ejecutar determinado comando con sus argumentos
- `continue`: salta a la siguiente iteración en fors, whiles, selects o untils
- `declare`: declaración de variables
- `dirs` : muestra la pila de directorios
- `disown`: quita los procesos dependientes del shell para que no dependan de este
- `do`, `done` : parte de las iteraciones
- `echo` : mostrar datos por salida
- `enable`: habilitar o deshabilitar los comandos builtin.
- `exec` : ejecuta un comando en lugar del proceso actual
- `exit`: termina el script o el shell
- `export`: convierte las variables en globales
- `fc` : editar comandos del historial
- `fg`: pasar comandos a primer plano
- `getopts`: para tomar los argumentos opcionales del script, al estilo de c
- `help` : muestra la ayuda
- `history`: el historial de comandos
- `if`: para crear estructuras condicionales
- `jobs`: muestra los procesos que tenemos en marcha
- `kill`: envío de señales a procesos.
- `let`: ejecución de operaciones aritméticas
- `local`: declaración de variables locales
- `logout`: salida del shell
- `popd`: saca un directorio de la pila de directorios
- `printf`: sacar datos formateados como en c
- `pwd`: muestra el directorio actual
- `pushd`: mete un directorio en la pila
- `read`: lee desde la entrada estándar
- `readonly`: previene que variables de shell sean sobrescritas
- `return`: para terminar funciones
- `select`: estructura de selección
- `set`: muestra variables del shell, y puede establecer muchas opciones
- `shopt`: establece o quita opciones de shell
- `shift`: desplaza posiciones en el shell
- `source`: similar al .
- `suspend`: suspende el shell actual, suele hacerse en el su
- `test`: para construir condicionales
- `time`: mide el tiempo de ejecución de un comandos
- `times`: muestra tiempos acumulados de procesos

- `trap`: para atrapar señales
- `true`: constante booleana
- `type`:distingue si un comando es builtin, función,alias, palabra clave,..
- `ulimit`: muestra los limites de recursos del sistema
- `umask`: muestra/establece permisos que sse deben aplicar a nuevos ficheros
- `unalias`: quita los alias
- `unset`: desestablece funciones o variables
- `until`: para crear iteraciones
- `wait`: detiene la ejecución a la espera que termine otro proceso
- `while`: para crear iteraciones.

5.Programación de scripts de shell

Introducción

Un script de shell no es más que un fichero de texto que contiene una serie de comandos del sistema linux además de los comandos que forman parte del shell (built-in) con los que podemos crear estructuras que facilitan la programación de scripts complejos.

Para poder ejecutar los ficheros deben ser ejecutables por tanto si no es ejecutable hay que forzar que lo sea:

```
linux~$ chmod u+x scriptshell.sh
```

o con números:

```
linux~$ chmod 755 scriptshell.sh
```

A partir de ahí ya podemos ejecutar el script invocando directamente el fichero. En los sistemas linux y en el shell bash2 debemos hacerlo especificando el directorio actual:

```
linux~$ ./scriptshell.sh
```

Si no ponemos ./ el script no se ejecutará y el sistema dirá que no lo encuentra.

Otra forma de ejecutar el script es usar el comando `source`. A través de este comando podemos ejecutar un script aunque no tenga permisos de ejecución.

```
linux~$ source scriptshell.sh
```

Si no ponemos ./ el script no se ejecutará y el sistema dirá que no lo encuentra.

Hola mundo

Este es el aspecto que tendría el script más básico posible:

```
#!/bin/bash
# Esto es un comentario
echo "Hola mundo"
```

Este sería similar pero usando una variable:

```
#!/bin/bash
# En la primera linea establezco con qué shell se debe ejecutar

# Se define una variable
SALUDO="Hola mundo"
```

```
echo -n "Este script te dice: "
echo ${SALUDO}
```

Variables

En el shell se pueden definir variables y por tanto dentro de los scripts también. Aquí no hay declaración de tipos pero sí que podemos definir variables que contienen números, cadenas, booleanos e incluso arrays.

Para declarar una variable basta con hacer:

```
VARIABLE=valor
```

La podemos exportar para que se convierta en una variable global.

```
export VARIABLE
```

A partir de que se crea la variable ya podemos acceder a ella a través de su nombre con el símbolo del dólar por delante.

```
echo $VARIABLE
```

Este es un ejemplo de uso de variables:

```
#!/bin/sh

# Muestra el uso de variables

# No existen los tipos

# definición de variables
una_variable=666
MI_NOMBRE="Richard"
NOMBRES="Iñigo Asier Sten Roberto Pello"
BOOLEANO=true

echo "Echemos un ojo a las variables "
echo "Un número: ${una_variable}"
echo "Un nombre ${MI_NOMBRE}"
echo "Un grupo de nombres: ${NOMBRES}"

# Al script se le pueden pasar argumentos. Para recogerlos
# hay que usar : ${número}  donde:
# ${0} : nombre del script
# ${1} : primer argumento
# ${2} : segundo argumento
# ...etc.
echo "Has invocado el script pasándome ${0} eta ${1} "
```

```
# Otras variables especiales
# $# : Número de argumentos
echo "Me has pasado $# argumentos"

# $@ : grupo de parámetros del script
echo IDa: ${!} y $@

# Variables de entorno
echo "Mi directorio actual: ${PWD} y mi usuario ${UID}"
```

---

arrays

Como ya se ha dicho, existe la posibilidad de declarar y usar arrays. Son algo limitados pero pueden resultar prácticos.

Este es un ejemplo de uso de arrays

```
#!/bin/bash

# Muestra el uso de arrays

# Podemos crear arrays de una dimensión
asociaciones[0]="Gruslin"
asociaciones[1]="Hackresi"
asociaciones[2]="NavarradotNET"
asociaciones[3]="Riberlug"

# Otra forma de definirlos
partidos=("UPN" "PSN" "CDN" "IUN" "Nafarroa BAI" "RCN" )
numeros=(15 23 45 42 23 1337 23 666 69)

echo ${asociaciones[0]} es una asociación, ${partidos[1]} un partido

#echo "Tamaño: $#asociaciones"
#echo "Tamaño: ${$#{partidos}}"
```

Otros usos

Muchas veces puede que nos interese tomar solamente parte del valor de una variable, o asignarle un valor por defecto. Este es un ejemplo de algunas cosas que podemos hacer con las variables:

```
#!/bin/bash

# Muestra el uso de variables
```

```
# Sacar parte del valor de una variable (substring)
NOMBRE="Navarrux Live edition"
echo ${NOMBRE} una parte ${NOMBRE:8} y otra ${NOMBRE:8:4}

# Valores por defecto.
# ${variable:-valorpordefecto}
SINVALOR=
echo "Variable SINVALOR: ${SINVALOR:-31337} eta ${VACIO:-'Miguel Indurain'}"
echo "La variable SINVALOR no tiene valor algun ${SINVALOR} "

# Esto es igual pero el valor queda establecido
# ${variable:=valorpordefecto}
echo "Variable SINVALOR: ${SINVALOR:=31337} eta ${VACIO:='Miguel Indurain'}"
echo "La variable SINVALOR no tiene valor algun ${SINVALOR} "

# Y lo contrario: si la variable SÍ tiene valor, se le pone otro
# ${variable:+valorpordefecto}
PROGRAMA='sniffit'
echo "valor de PROGRAMA: ${PROGRAMA:+'tcpdump'} y ahora ${PROGRAMA}"

# Si la variable está sin definir o vacía se muestra un error. En caso
# contrario se le asigna un valor por defecto
# ${variable:?valorpordefecto}
echo "Valor de: ${EJEMPLO:?'Roberto'} y luego {EJEMPLO}"

# Otros
# ${!prefijo*} : nos devuelve una lista de las variables que comienzan
# con determinado prefijo.
# Podemos probar con el for:
for i in ${!P*};do echo $i ;done
```

Operaciones

Ya que tenemos variables, que menos que poder operar con ellas de alguna forma, Al igual que en cualquier otro lenguaje de programación disponemos de operadores aritméticos y lógicos, aunque su uso no es tan simple.

Aritméticas

Este es un ejemplo del uso de operaciones aritméticas

```
#!/bin/bash

# -, +, *, /, %, **, variable++, variable--, --variable, ++variable
# == : igualdad
```

```
# != : desigualdad

# Pruebas
VALOR1=23
VALOR2=45

# Para las operaciones puede usarse expr o []

RESULTADO=`expr ${VALOR1} + ${VALOR2}`
echo "Resultado: ${RESULTADO}"

RESULTADO=`expr ${VALOR1} + ${VALOR2} + 3`
echo "Resultado: ${RESULTADO}"


VALOR1=5
VALOR2=4
echo "${VALOR1} y ${VALOR2}"

RESULTADO=$[${VALOR1} + ${VALOR2} + 2]
echo "Ahora: ${VALOR1} + ${VALOR2} + 2 = ${RESULTADO}"


RESULTADO=$[${VALOR1} + $[${VALOR2} * 3]]
echo "Ahora: ${VALOR1} + ${VALOR2} * 3 = ${RESULTADO}"
```

Lógicas

Este es un ejemplo de uso de operaciones lógicas

```
#!/bin/bash

## operaciones lógicas

#  && : and
#  || : or
#  !  : not

booleano=true

# Si la variable booleano es true veremos por pantalla "OK"
$booleano && echo "OK" || echo "no es true"

otrobool=!${booleano}
```

```
test ${otrobool} || echo "Ahora es falso"


# Mediante && podemos encadenar comandos
#who && ps -axf && echo "OK"


## comparaciones : -eq, -ne, -lt, -le, -gt, or -ge

valor=6

test $valor -le 6  && echo "Se cumple"

# Asignaciones

nuevo=${valor}

test  ${nuevo} -eq ${valor}  && echo "Son los mismo"

echo "Ahora ${nuevo} es lo mismo que ${valor}"
```

Condicionales

En el shell podemos crear las habituales estructuras condicionales if o if-else.

Esta sería la forma de if:

```
if condición; then
     operaciones
fi
```

o también:

```
if condición
then
     operaciones
fi
```

Esta sería la forma de el if-else

```
if condición
then
     operaciones
else
```

```
     operaciones
fi
```

Y también tenemos el if-elseif...-else

```
if condición
then
     operaciones
elif condición
then
     operaciones
else
     operaciones
fi
```

Este es un ejemplo de uso de condicionales if y test

```
#!/bin/bash

# Condicionales

VARIABLE=45

if [ ${VARIABLE} -gt 0 ]; then
 echo "${VARIABLE} es mayor que 0"
fi

# Podemos meter el then en la siguiente linea
if [ -e /etc/shadow ]
then
    echo "OK, parece que tienes un sistema con shadow pass"
fi


## Estructura if-else

OTRA=-23

if [ ${OTRA} -lt 0 ]
then
 echo "${OTRA} es menor que 0"
else
 echo "${OTRA} es mayor que 0";
```

```
fi

## Estructura if-elseif


# Vamos a usar "read" para pedirle datos al usuario

echo -n "Mete un valor: "
read VALOR1
echo -n "Mete otro valor: "
read VALOR2

echo "Has introducido los valores ${VALOR1} y ${VALOR2} "

if [ ${VALOR1} -gt ${VALOR2} ]
then
   echo "${VALOR1} es mayor que ${VALOR2}"
elif [ ${VALOR1} -lt ${VALOR2} ]
then
   echo "${VALOR1} es menor que ${VALOR2}"
else
   echo "${VALOR1} y ${VALOR2} son iguales"
fi


## Estructura test
# Ejecutar operacion si no se cumple la condición
# test  condición || operacion
# Ejecutar operación si se cumple
# test  condición && operacion

# Este test crea un fichero si no existe.
test  -f './fichero.txt' || touch fichero.txt
```

Comprobaciones que podemos hacer:
En las condicionales podemos hacer una serie de comprobaciones con ficheros, las mostramos en orden alfabético:

- `-a fichero` : verdadero si el fichero existe
- `-e fichero` : lo mismo
- `-b fichero` : verdadero si el fichero existe y tiene algún bloque especial
- `-c fichero` : verdadero si el fichero existe y es del tipo carácter (suelen ser dispositivos)
- `-d fichero` : verdadero si el fichero existe y es un directorio.

- `-f fichero` : verdadero si el fichero existe y es un fichero común.
- `-g fichero` : verdadero si el fichero existe y tiene el bit setGruopID establecido
- `-h fichero` : verdadero si el fichero existe y es un enlace símbolico
- `-k fichero` : verdadero si el fichero existe y tiene el sticky bit establecido
- `-p fichero` : verdadero si el fichero existe y es una tubería con nombre.
- `-r fichero` : verdadero si el fichero existe y tiene permisos de lectura.
- `-s fichero` : verdadero si el fichero existe y es mayor que 0.
- `-u fichero` : verdadero si el fichero existe y tiene el bit setUID establecido.
- `-w fichero` : verdadero si el fichero existe y tiene permisos de escritura.
- `-x fichero` : verdadero si el fichero existe y tiene permisos de ejecución .
- `-O fichero` : verdadero si el fichero existe y el EUID es de nuetro usuario.
- `-G fichero` : verdadero si el fichero existe y el EGID es de nuestro grupo.
- `-L fichero` : verdadero si el fichero existe y es un enalce simbólico.
- `-S fichero` : verdadero si el fichero existe y es un socket.
- `-N fichero` : verdadero si el fichero existe y ha cambiado tras la última lectura.
- `-t descriptor` : verdadero si en un descriptor de fichero abierto y un terminal

A partir de ahí podemos hacer las comprobaciones tan complejas como haga falta:

```
#!/bin/sh
# script para comprobar si un fichero existe. Espera un parámetro
if [ -e $1 ] && [ -f $1 ]
then
echo "OK, el fichero existe "
else echo "NO existe"
fi

# Para comprobar la negación usaríamos el símbolo: !
if [ ! -e $1 ]
then
echo “No existe”
fi
```

Otros test que podemos hacer comparando ficheros:

- `fichero1 -nt fichero2` : verdadero si fichero1 es más reciente que fichero2 o si el fichero1 existe y el otro no.
- `fichero1 -ot fichero2`: verdadero si fichero1 es más viejo que fichero2 o si el fichero1 existe y el otro no.
- `fichero1 -ef fichero2`: verdadero si los dos comparten el mismo número de inodo y dispositivo.

case

El case o el switch-case tan típico del lenguaje C también está disponible aquí. Esta sería su forma:

```
case variable in
  valor1)
     operaciones_si_variable=valor1
     ;;
  valor2)
     operaciones_si_variable=varlo2
     ;;
       *)
     operaciones_en_cualquier_otro_caso
esac
```

Este es un ejemplo de uso de la estructura case

```
#!/bin/bash


NOMBRE=""

echo -n "Dame un nombre: "
read NOMBRE

case ${NOMBRE} in
  iñigo)
      echo "${NOMBRE} dice:  me dedico a Navarrux"
      ;;
  sten)
      echo "${NOMBRE} dice: tengo un blog friki"
```

```
      ;;
  asier)
      echo "${NOMBRE}> dice: .NET me gusta"
      ;;
  pello)
      echo "${NOMBRE}> dice: el shell mola"
      ;;
   *)
      echo "A ${NOMBRE} no lo conozco"
esac
```

Iteraciones

También tenemos las clásicas iteraciones for y while además de otras, siempre que necesitemos realizar tareas repetitivas, recorrer arrays o resultados de comandos, etc...

for

Se puede hacer un for de distintas maneras. El más simple tiene esta forma:

```
for var in lista

do

 operaciones

done
```

Aunque se pueden crear los típicos for que se inicializan con un valor y se hacen ejecutar x veces. Mejor verlo en código:

Este es un ejemplo de uso de la iteración for:

```
#!/bin/bash


## un for simple

echo "FOR simple: "
for i in a b c d e f g h i
do
 echo  "letra: $i"
done


## for para recorrer array
NOMBRES="Iñigo Sten Asier Pello Roberto"
echo "FOR simple para recorrer un array: "
```

```
echo "Participantes en la 4party: "

for i in ${NOMBRES}
do
      echo ${i}
done


## for con el resultado de un comando
echo "FOR con el resultado de un comando"

for i in `cat direcciones.txt`
do
      echo ${i}
done

## for con un grupo de ficheros
echo "FOR con ficheros"

for i in *.sh
do
      ls -lh ${i}
done


## for clásico con un índice
echo "FOR clásico "

for (( cont=0 ; ${cont} < 10 ; cont=`expr $cont + 1` ))
do
      echo "Ahora valgo> ${cont}"
done
```

---

while

Muchas veces nos interesa una iteración sin un número de vueltas fijo, que simplemente dependa de una condición. Eso lo podemos conseguir con el while, cuya forma resumida es esta:

```
while condición
do
      operaciones
done
```

Este es un ejemplo de uso de la iteración while:

```
#!/bin/bash


## Estructura while
echo "WHILE en marcha"

cont=0

# Un bucle que terminará cuando $cont sea mayor que 100
while (test ${cont} -lt 100)
do
   cont=`expr $cont + 10`
   echo "Valor del contador: ${cont}"
done

echo "Valor final del contador: ${cont}"

## Un while infinito
# while true; do comando; done
```

until

Until es similar a while, salvo que su ejecución se detiene de forma inversa. Cuando la condición resulta falsa, termina el bucle. Esta es su forma:

```
until condición

do

     operaciones

done
```

Este es un ejemplo de uso de la iteración until:

```
#!/bin/bash

## El bucle until
# Un bucle until se ejecuta hasta que el test resulte falso

echo "Estructura until"

cont=15

until (test ${cont} -lt 0)
do
   cont=`expr $cont - 1`
```

```
   echo "Valor del contador: ${cont}"
done

echo "Valor final del contador: ${cont}"
```

---

select

Mediante select podemos crear menús de selección de manera muy cómoda. Podemos definir un array con todas las opciones y select las mostrará por nosotros. Esta es su forma general:

```
select variable in lista

do

     operaciones

done
```

Este es un ejemplo de uso de la iteración select

```
#!/bin/bash

## Estructura select

# El select es similar al choice de msdos
# sirve para crear menus de seleccion

echo "Select de distros"

DISTROS="Debian Ubuntu Navarrux Gentoo Mandriva"

echo "Selecciona la mejor opción por favor: ${resultado}"

select resultado in ${DISTROS}
do
  # Si la longitud de cadena de resultado es > 0 ya está seleccionado
 (test ${#resultado} -ne 0 ) && break

 echo "Selecciona la mejor opción por favor: ${resultado}"

done

echo "Sistema seleccionado: [${resultado}] longitud de cadena: ${#resultado}"
```

Funciones

Para estructurar mejor el código, dividir las partes complicadas e incluso reutilizar funcionalidades podemos crear. Las funciones de shell pueden recibir parámetros

con las variables especiales $1, $2, ...

Aquí vemos varios ejemplos de funciones

```
#!/bin/bash

## funciones

# Antes que nada hay que definir las funciones, si no daría error
# las funciones toman los parámetros con $numero, como el script

# variable
RESULTADO=0

# Una función que muestra valores por pantalla
muestrapantalla () {
 echo "Valores: $0> $1 y $2 y $3"
}

# Puede usarse return pero solo para devolver números
sumame () {
      echo "Estamos en la función: ${FUNCNAME}"
      echo "Parametros: $1 y $2"
      RESULTADO=`expr ${1} + ${2}`
      return 0
}

# Es posible la función recursiva
boom () {
      echo "No ejecutes esto... "
      boom
}

# La llamada de se puede hacer así, sin paréntesis
muestrapantalla 3 4 "epa"

# Llamamos a la función y si devuelve 0 es correcto.
sumame 45 67 && echo "OK" || echo "Ocurrió un error"

echo "Resultado: ${RESULTADO} $!"
```

Librerías

A la hora de programar scripts de shell no es que existan herramientas para declarar librerías, pero se pueden simular de cierta manera metiendo código de funciones en ficheros de scripts separados.

Este sería un ejemplo de uso de librerías, usando un fichero con las funciones

mostradas anteriormente:

```
#!/bin/bash

# No es que existan las librerías pero se puede simular
# algo similar

# Esta es la forma de importar funciones
source libreria.sh


muestrapantalla 69 123 "epa"

sumame 1337 3389 && echo "OK" || echo "Ocurrió un error"

echo "Resultado: ${RESULTADO} $!"
```

Señales

Las señales son un método de los sistemas linux para intercomunicar procesos. Las señales más frecuentes usadas por los usuarios son las de cancelar y matar procesos. Pero hay más y los procesos linux son capaces de capturar las señales para modificar su comportamiento.

Desde un script de shell también se pueden capturar las señales.

Este sería un simple ejemplo de la captura de señales:

```
#!/bin/bash

# señales

funcion () {
  echo "Se ha recibido una señal: ${FUNCNAME} ${0}"
  #exit
}

# Lo primero es establecer que señales se atraparán. Lo hacemos con trap
# Con esto evitaremos que se haga caso a Ctrl-C CTRL-Z
# trap ":" INT QUIT TSTP

# Esto es similar pero al recibir la señal dirigimos la ejecución a la función
trap "funcion" INT QUIT TSTP

# Un bucle sin fin para probar
while true
do
  sleep 2
```

```
  echo "ufff qué sueño..."
done
```

Colores

Usando la opción -e del comando echo podemos cambiar los colores de la terminal. Se puede hacer de forma controlada, estableciendo un color y restableciendo el color anterior para cambiar colores puntualmente. Para los colores se usan los códigos ANSI.

Este sería un ejemplo del uso de los códigos ANSI:

```
#!/bin/bash

## Los colores ANSI
# para darle color a los scripts debemos usar los código ANSI
# junto "con echo -e". Para meter el carácter especial escape
# usamos \033

# Ejem: \033[0m : volver al color por defecto
# \033[40m: color de fondo negro
# \033[40m\033[32m: fondo negro primer plano verde

# Esta es la muestra de colores, ejecútalo para ver cómo queda
echo -e "\033[40m\033[37m  Blanco        \033[0m"
echo -e "\033[40m\033[1;37m  Gris claro   \033[0m"
echo -e "\033[40m\033[30m  Negro        \033[0m (esto es negro)"
echo -e "\033[40m\033[1;30m  Gris         \033[0m"
echo -e "\033[40m\033[31m  Rojo          \033[0m"
echo -e "\033[40m\033[1;31m  Rojo claro    \033[0m"
echo -e "\033[40m\033[32m  Verde       \033[0m"
echo -e "\033[40m\033[1;32m  Verde claro  \033[0m"
echo -e "\033[40m\033[33m  Marrón        \033[0m"
echo -e "\033[40m\033[1;33m  Amarillo       \033[0m"
echo -e "\033[40m\033[34m  Azul         \033[0m"
echo -e "\033[40m\033[1;34m  Azul claro   \033[0m"
echo -e "\033[40m\033[35m  Purpura       \033[0m"
echo -e "\033[40m\033[1;35m  Rosa         \033[0m"
echo -e "\033[40m\033[36m  Cyan         \033[0m"
echo -e "\033[40m\033[1;36m  Cyan claro   \033[0m"

# Se pueden poner fondos de otro color:
echo -e "\033[42m\033[31m  Navarrux v1.0        \033[0m"

echo -e "\033[40m\033[4;33m  Amarillo         \033[0m"
```

6.Bibliografía, referencias y enlaces

Bibliografía

Learning The Bash Shell – O'Reilly

Bash Quick Reference - O'Reilly

Linux Shell Scripting With Bash - Sams

Unix Shell Programming 3Rd - Sams

Referencias

man bash: manual de sistema de bash,

Enlaces

http://es.tldp.org/

http://www.navarrux.org

http://aula0.cuatrovientos.org

http://www.cuatrovientos.org

¡Saludos a Iñigo Tejedor!