RedTauros ltda

http://www.redtauros.com

Índice de contenido

EL COMANDO APT-GET

apt-get es la herramienta que utiliza Debian y sus derivadas (Ubuntu incluida), para gestionar los paquetes instalables disponibles en los repositorios y aunque tenemos a nuestra disposición herramientas gráficas que nos facilitan las cosas, nunca está de más saber lo que podemos hacer con apt-get desde una terminal:

Comandos

- **apt-get update** : Actualiza el listado de paquetes disponibles.
- **apt-get check** : Comprueba que todo ha ido bien tras la utilización de apt-get update.
- **apt-get install** *lista_de_paquetes* : Instala los programas deseados.
- **apt-get -reinstall install** *paquete* : Reinstala un programa.
- **apt-get upgrade** : Actualiza los paquetes ya instalados.
- **apt-get dist-upgrade** : Actualiza toda la distribución sin retener paquetes.
- **apt-get remove** *lista_de_paquetes* : Desinstala un paquete.
- **apt-get -purge remove** *lista_de_paquetes* : Desinstala un paquete y elimina los archivos de configuración.
- **apt-get -f install** : Resuelve problemas con dependencias y paquetes rotos. Puede ser necesario reconfigurar dpkg con: **sudo dpkg --configure -a**
- **apt-get clean** : Para limpiar los paquetes descargados e instalados.
- **apt-get autoclean** : Para limpiar los paquetes viejos que ya no se usan.
- **apt-cache search** *nombre paquete* : Para buscar un paquete determinado.
- **apt-get source paquete** : Descarga archivos fuente.

ARCHIVOS DE CONFIGURACIÓN (/ETC)

GNU/Linux es increíblemente fácil de configurar, no existen bases de datos raras, ni registros, ni directorios regados por aquí y por allá con archivos extraños, ni 'dll hell', no archivos con terminación .ini o .bat o algo más, etc. Casi todo lo configurable (por no decir todo) lo encuentras en el directorio de configuración etc, y todos los archivos configurables de este directorio no son más que simples archivos de texto ASCII, editables desde cualquier editor, más simple no es posible. Pero este directorio tiene decenas de archivos y subdirectorios.

Por cierto, 'etc' (que los norteamericanos pronuncian EtSee 'etsi') aparenta querer decir 'etcetera', pero según literatura de libros viejos de Unix de los 80's, etc son realmente las iniciales de "Extended Tool Chest", no lo podría asegurar pero parece ser que ese es su verdadero significado del viejo y venerable directorio 'etc' presente en todos los sistemas Linux y Unix. Aunque hay otro significado que podría ser el mas acertado y es “Edit This Carefully”, que traduciría algo como “edite esto con cuidado”.

Donde se encuentre 'Dir' indica directorio, todos los demás son archivos.

Archivos de configuración en /etc

/etc/aliases	Permite agregar alias (nicks) a nombres reales de usuarios de correo electrónico.

/etc/bashrc	Funciones y alias disponibles para todos los usuarios, variables de entorno globales en /etc/profile.
/etc/cron.d	Dir, archivos de cron personalizados para programas específicos.
/etc/cron.daily	Dir, scripts de usuarios o de programas específicos que se ejcutan cada día, según lo definido en crontab.
/etc/cron.hourly	Dir, scripts de usuarios o de programas específicos que se ejcutan cada hora, según lo definido en crontab.
/etc/cron.monthly	Dir, scripts de usuarios o de programas específicos que se ejcutan cada mes, según lo definido en crontab.
/etc/cron.weekly	Dir, scripts de usuarios o de programas específicos que se ejcutan cada semana, según lo definido en crontab.
/etc/crontab	Controla archivos de cron para usuarios individuales o para el usuario root.
/etc/fedora-release	Sustitue 'fedora' por el nombre de tu distro para ver la versión específica de tu distribución Linux.
/etc/exports	Definición de directorios a compartir a través del sistema de archivos en red NFS.
/etc/filesystems	Se usa para probar el orden de sistemas de archivos cuando se monta un dispositivo con la opción auto.
/etc/fstab	Lista los sistemas de archivos montados automáticamente al arranque del sistema
/etc/group	Almacena la información de los grupos del sistema, complemento de /etc/passwd
/etc/gshadow	Guarda las contraseñas de los grupos asi como información de la caducidad de la misma, similar a /etc/shadow
/etc/host.conf	Indica como en que orden se resuelven los nombres de equipo o de dominio.
/etc/hosts	Define nombres de equipos igualándolos con sus direcciones IP.
/etc/hosts.allow	Define un formato de acceso o lista de control de acceso de que equipos pueden ingresar al sistema.
/etc/hosts.deny	Define un formato de acceso o lista de control de acceso de que equipos no pueden ingresar al sistema.
/etc/inittab	Archivo de configuración para el comando init, determina el nivel de ejecución del sistema y define scripts de arranque.
/etc/issue	Mensaje de bienvenida para todos las consolas antes del login.
/etc/login.defs	Controla la configuración del login de usuarios (contraseña, caducidad, etc.) en sistemas que usan /etc/shadow
/etc/logrotate.conf	Configura los parámetros del programa logrotate que a la vez administra archivos de bitácora (logfiles).
/etc/mtab	Archivo dinámico que contiene una lista de los sistemas de archivos montados actualmente. Inicializado por init y actualizado por mount.
/etc/motd	"Message Of The Day", mensaje que aparece a todos los usuarios después de loguearse a una terminal.
/etc/passwd	La base de datos de usuarios del sistema, nombre, directorio de inicio, id del usuario, etc. Se complementa con las contraseñas almacenadas en /etc/shadow
/etc/printcap	Archivo de configuración para las impresoras.
/etc/profile	Variables de entorno globales a todos los usuarios. Funciones y alias van en /etc/bashrc
/etc/rc.d	Dir, directorio que contiene los scripts de arranque del sistema y los directorios de los niveles de ejecucción.
/etc/rc.d/init.d	Dir, scripts de arranque/detener de los diferentes programas

	servidores del sistema. en algunas distros esta en /etc/init.d
/etc/rc.d/rc.local	Último script que se ejecuta al arranque del sistema, es el más adecuado para agregar nuestros propios script de arranque.
/etc/rc.d/rc0.d	Dir, scripts de arranque(Start)/detener(Kill) cuando se ingresa al nivel de ejecución 0 (apagado del equipo).
/etc/rc.d/rc1.d	Dir, scripts de arranque(Start)/detener(Kill) cuando se ingresa al nivel de ejecución 1 (monousuario, single user).
/etc/rc.d/rc2.d	Dir, scripts de arranque(Start)/detener(Kill) cuando se ingresa al nivel de ejecución 2 (multiusuario).
/etc/rc.d/rc3.d	Dir, scripts de arranque(Start)/detener(Kill) cuando se ingresa al nivel de ejecución 3 (red completa, multiusuario).
/etc/rc.d/rc4.d	Dir, scripts de arranque(Start)/detener(Kill) cuando se ingresa al nivel de ejecución 4 (personalizado).
/etc/rc.d/rc5.d	Dir, scripts de arranque(Start)/detener(Kill) cuando se ingresa al nivel de ejecución 5 (modo gráfico X11, red completa, multiusuario).
/etc/rc.d/rc6.d	Dir, scripts de arranque(Start)/detener(Kill) cuando se ingresa al nivel de ejecución 6 (reinicio del equipo).
/etc/resolv.conf	Configura la(s) dirección(es) del servidor de nombres de domino que resuelve para el equipo.
/etc/securetty	Identifica las terminales en las que el usuario root puede loguearse.
/etc/services	Lista de los servicios de red (tcp y udp) según la última lista de la iana.org
/etc/shells	Lista de los shell (línea de comandos) confiables.
/etc/shadow	Complemento de /etc/passwd, archivo donde se guarda la contraseña encriptada y demás datos de la misma de los usuarios del sistema.
/etc/sudoers	Lista de usuarios con privilegios especiales de root y los comandos que pueden ejecutar.
/etc/sysconfig	Dir, directorio donde se almacenan archivos de configuración relativos al equipo, teclado, mouse, red, etc.
/etc/ " /clock	Permite definir la zona horaria y otros parámetros de la fecha y hora.
/etc/ " /i18n	Parámetros LC (locale) y otros de internacionalización como sistema de medida, de moneda, código de teléfono, etc.
/etc/ " /init	Variables de control de la forma en que inicia el sistema.
/etc/ " /iptables	Iptables toma por default este archivo para cargar sus reglas al arranque del sistema.
/etc/ " /network	variables de configuración global de parámetros de red.
/etc/ " /networking/devices	Dir, directorio que contiene la configuración de los dispositivos de red.
/etc/ " / " / " /ifcfg-eth0	Cada dispositivo (eth0, eth1, etc.) de red tiene su archivo de variables de configuración.
/etc/sysctl.conf	Variables de configuración del kernel.
/etc/syslog.conf	Control y configuración sobre la bitacorización de eventos del sistema.
/etc/termcap	Configuración de los atributos de la terminal o shell.
/etc/version	Generalmente el número de versión de la distro.

Es importante entender que no todos los archivos aqui mostrados existen en todas las distribuciones de Linux, puede haber y de hecho las hay importantes diferencias, pero en su gran mayoría esta guía aplica.

EL PROCESO INIT

El proceso de arranque init de Sys V es el primer proceso que se ejecuta en el sistema, es el más importante, del que dependen el resto de todos los demás procesos. En el arranque de GNU Linux, el núcleo ejecuta init. Este programa, ahora como proceso, cargará los subprocesos necesarios para la puesta en marcha del sistema. Cuando init haya terminado de cargarse vaciará el subdirectorio /tmp y lanzará a getty que es el encargado de permitir a los usuarios hacer login en el sistema.

Los niveles de ejecución (también generalmente conocidos por su nombre en inglés, runlevel) determinan los servicios que tendremos disponibles en cada uno de ellos. Es una forma de tener diferentes modos de trabajo, cada uno de ellos con distintas características bien definidas, en función del tipo de tarea a que estén orientados.

Existen ocho niveles de ejecución: los nombres de los siete primeros son los números que van del 0 al 6, más un octavo cuyo nombre es la letra S (tiene un alias con la letra s para evitar problemas con la sensibilidad al caso), este runlevel, en realidad, es igual a el nº 1.

Los niveles de ejecución son tal como siguen:

0: (Detener el sistema).

1: (modo en Mono usuario, sin soporte de red).

2: (modo en Multiusuario, sin soporte de red).

3: (Modo multiusuario completo).

4: (Sin uso. Recomendado para pruebas).

5: (Multiusuario completo en entorno gráfico).

6: (Reinicio del sistema).

Los niveles 0, 1 y 6 son comunes en todas las distribuciones, el resto puede cambiar dependiendo de cada una de las que estemos usando.

Init necesita un fichero de configuración para saber exactamente lo que tiene que hacer. Este fichero es /etc/inittab y contiene información sobre el runlevel a ejecutar por defecto, previsión sobre lo que hacer ante determinadas situaciones, así como una descripción de qué procesos se han de iniciar en la carga y durante la operación normal del sistema operativo.

Las entradas del fichero /etc/inittab tienen el siguiente formato:

id:niveles_ejecución:acción:proceso

id: Es la secuencia única de 1 a 4 caracteres que identifican la entrada de inittab.

niveles_ejecución: Lista de niveles de ejecución para los que se llevarán a cabo las acciones definidas a continuación en la misma línea.

acción: La acción que será llevada a cabo.

proceso: El proceso a ejecutar.

Para que una línea sirva para varios niveles de ejecución, el campo niveles_ejecución tiene que incluirlos. Por ejemplo, 135 indica que el proceso se iniciará en los niveles 1, 3 y 5. Cuando se cambia de un nivel de ejecución a otro, los procesos en ejecución que no estén definidos en el nuevo nivel se matan. Para ver un ejemplo de esta sintaxis edita el fichero initab con nano o vi tal cual vimos como se hacía en capítulos pasados y examina su contenido cuidadosamente.

Mientras no tengas claro el sistema de runlevel y necesites modificar algo, mi consejo es que no lo toques, simplemente examínalo por curiosidad y para tu información; este proceso casi nunca será necesario hacerlo.

Las acciones que podemos definir, más habitualmente, en el campo acción son:

initdefault: Especifica el nivel de ejecución por defecto al arrancar el sistema. El campo proceso se ignora.

Respawn: El proceso se reiniciará cuando termine.

once: El proceso se ejecutará una sola vez cuando se entre en el nivel de ejecución especificado.

wait: El proceso se iniciará una vez cuando se entre en el nivel de ejecución e init esperará a su terminación.

boot: El proceso se ejecutará durante el arranque del sistema. El campo niveles_ejecución se ignora.

bootwait: El proceso se ejecutará durante el arranque del sistema, mientras init espera su terminación. El campo niveles_ejecución se ignora.

sysinit: El proceso se ejecutará durante el arranque del sistema, antes que cualquier entrada boot o bootwait. El campo niveles_ejecución se ignora.

powerwait: El proceso se ejecutará si init recibe una señal SIGPWR, que indica algún problema con la alimentación eléctrica. Init esperará que el proceso

termine.

powerfail: Como powerwait, excepto que init no espera a que termine el proceso.

powerokwait: El proceso se ejecutará si init recibe la señal SIGPWR, con la condición de que haya un fichero llamado /etc/powerstatus que contenga la palabra

OK. Esto significará que se ha restablecido la alimentación eléctrica.

ctrlaltdel: Especifica qué proceso se ejecutará al pulsar la combinación de teclas Control+Alt+Suprimir. que normalmente será reiniciar la máquina.

Hay un directorio para cada nivel de ejecución. En cada uno de los directorios figuran las aplicaciones que se inician o se paran con ese nivel de ejecución. En realidad no están las aplicaciones como tales, sino que tenemos allí enlaces simbólicos hacia ellas. Por ejemplo, para el nivel de ejecución 2, las aplicaciones están contenidas en el directorio: /etc/rc2.d, y así para todos los demás niveles. Si queremos, por tanto, que una aplicación se inicie al arrancar en un nivel determinado, hay que crear un enlace simbólico en el directorio correspondiente al nivel de ejecución deseado que apunte al script encargado de arrancar la aplicación que estará en /etc/init.d que, como supongo acabaréis de recordar, es el directorio donde se situaban los scripts lanzadores de los daemons o demonios.

Si Observamos atentamente el nombre de los enlaces simbólicos de cada uno de los directorios de nivel de ejecución, Podremos advertir que cada uno de ellos tendrá el nombre del script al que está asociado. Los que empiecen con una S (S

de start) están indicando que el servicio se iniciará, y los que comiencen con una K (K de kill), están indicando que el servicio se detendrá. El número que suele aparecer es simplemente una facilidad para poder ordenarlos y que no tiene mayor relevancia.

Lo bueno de este sistema es que, en primer lugar, no se han de repetir los scripts en cada directorio de runlevel, si no que permanecerán en un único lugar bien definido, el directorio init.d, y en segundo lugar, la modificación a realizar si lanzamos un servicio o no, en un runlevel determinado, es tan sencilla como cambiar el nombre del enlace al servicio en cuestión. Si queremos que se inicie, por tanto, bastará con asegurarse de que su nombre comience por una S y en caso contrario, osea que en ese nivel de ejecución no se ofrezca el

servicio, pondremos el nombre empezando por una K. Asimismo, otra ventaja resultante
de este método de funcionamiento es el control que tenemos sobre los servicios del sistema, que es independiente del estado en el que estén. Podremos lanzarlos, detenerlos, reiniciarlos, etc. sobre la marcha, sin necesidad de reiniciar la máquina.

Veamos un ejemplo: si deseásemos lanzar la línea braille en el arranque, imaginando que en nuestro ejemplo El arranque por omisión estuviera en el nivel 2, y lanzar al demonio BRLTTY que controla las líneas braille en el sistema:

Primero decidiremos qué nivel de ejecución queremos utilizar para ello. Si fuera el nivel 2 habría que crear el enlace simbólico en rc2.d el enlace podría ser algo parecido a lo siguiente: sXbrltty donde la X deberemos substituirla por el número de órden que queramos.

Desde el directorio /etc/rc2.d la orden sería: ln –s ../init.d/brltty sXbrltty

Uno de los scripts más importantes en el arranque del sistema es /etc/rc.d/rc.sysinit. y es el primer script que init ejecuta. En él están definidas funciones
importantes como pueden ser: el inicio y activación del espacio de intercambio. (swap), la configuración de la red, la especificación de las variables del sistema, la comprobación y montaje de los sistemas de archivos, la inicialización de puertos serie, la carga de los módulos del kernel, el establecimiento
de las cuotas para cada usuario, el ajuste del reloj del sistema, etc.

El último script en ejecutarse es /etc/rc.d/rc.local. En este fichero se podrán poner inicializaciones especificas del sistema, aunque su propósito inicial es controlar los servicios de red.

init como orden:

Además de todo lo que hemos visto con respecto al proceso init, también podremos ejecutarlo como una orden desde línea de comandos con alguna de las siguientes opciones:

0, 1, 2, 3, 4, 5, 6: Para cambiar al nivel de ejecución especificado tal cual lo explicamos un poco más arriba.

q: Si queremos que init relea el fichero /etc/inittab.

s: Entra en modo monousuario.

u: Reejecuta init respetando el estado actual. No se relee el fichero /etc/inittab.

ACTUALIZANDO DEBIAN/UBUNTU SIN SATURAR LA RED. REPOSITORIO LOCAL.

Muchas veces, tenemos en nuestra empresa u oficina, varios equipos con Linux Ubuntu. Y pensamos en el consumo de tiempo y de ancho de banda, cuando cada uno de los equipos se actualizan o instalamos un software. Seria muy bueno tener un repositorio local, en nuestra oficina o empresa, en donde todos los equipos se actualicen e instalen software desde el (sin ocupar el Internet). Ademas de que este equipo en donde tenemos el repositorio se actualice solo diariamente.

Una vez que tengamos nuestro PC designado a este fin instalamos los siguientes paquetes:

```
sudo apt-get install apt-mirror apache2
```

El paquete apt-mirror es quien nos ayudará en este proyecto. Vamos a programarlo para que todos los días a cierta hora comience el proceso de descarga de actualizaciones. Para ello editamos el fichero

```
sudo gedit /etc/cron.d/apt-mirror
```

quitando el # en la ultima linea y definiendo la hora en la cual comenzará. Por ejemplo a las 7 de la mañana.

Ahora queda encargarnos de definir el medio de publicación, por eso hemos instalado apache. Lo resolvemos definiendo el siguiente enlace simbólico.

```
sudo ln -s /var/spool/apt-mirror/mirror/archive.ubuntu.com/ubuntu/ /var/www/
```

Y arrancamos todo:

```
sudo apt-mirror
```

Esto nos descargará una imagen de todos los paquetes disponibles en servidor de repositorios. Si quereis limitar esto o definir exactamente que tipo de actualización descargar. Podéis editarlo con

```
sudo gedit /etc/apt/mirror.list
```

Ahora solo nos queda, mientras el servidor descarga las actualizaciones configurar el resto de equipos.

Para ello editamos, en cada uno de ellos el sources.list con

```
sudo gedit /etc/apt/sources.list
```

comentamos añadiendo # a todas las lineas y por ultimo añadimos lo siguiente:

```
deb http://192.168.0.10/ubuntu/ maverick main restricted
deb http://192.168.0.10/ubuntu/ maverick-updates main restricted
deb http://192.168.0.10/ubuntu/ maverick-security main restricted
```

Donde 192.168.0.10 es la dirección del servidor que hemos definido en los pasos anteriores. Guardamos, y actualizamos con:

```
sudo apt-get update
```

Listo, ya tenemos nuestro repositorio local funcionando.

Nota:
Si alguna vez veremos que nuestro apt-mirror nos da un error:

```
apt-mirror is already running , exiting at /usr/bin/apt-mirror line 187.
```

Para arreglarlo basta con borrar un archivo:

```
rm /var/spool/apt-mirror/var/apt-mirror.lock
```

Hay que tener en cuenta que bajar todo un repositorio ocupa un espacio considerable, de mas o menos 80 GB.

CONEXIONES REMOTAS POR CONSOLA

SSH (*Secure Shell* o intérprete de órdenes seguro) es el nombre de un protocolo y del programa que lo implementa, y sirve para acceder a máquinas remotas a través de una red. Permite controlar un ordenador remoto mediante un intérprete de comandos, y redirigir el tráfico de un servidor para poder ejecutar programas gráficos si tenemos un servidor Unix con las X instaladas y correctamente configuradas.

OpenSSH (*Open Secure Shell*) es un conjunto de aplicaciones que permiten realizar comunicaciones cifradas a través de una red, usando el protocolo SSH. Fue creado como una ***alternativa libre y abierta al programa Secure Shell, que es software propietario.***

En este pequeño tutorial vamos a instalarlo en un equipo para poder controlarlo de forma remota.

Instalamos, por parte del servidor el paquete ***openssh-server***, aunque es recomendable instalar directamente el paquete completo con el siguiente comando:

```
sudo apt-get install ssh
```

Una vez instalado (tarda muy poco) podemos hacer la prueba de forma local en el ordenador servidor, o bien desde otra máquina con el siguiente comando:

```
sudo ssh usuario@ip_del_servidor
```

Y listo, una vez ejecutado ya podemos administrar de forma remota nuestro servidor sin ningún tipo de periféricos.

Para cerrar la conexión SSH basta con escribir el comando:

```
quit
```

Por lo demás, es como si realmente estuvieras dentro del terminal de Ubuntu, por lo tanto los comandos son exactamente los mismos.

Y esto es todo, ya tenemos instalado nuestro servidor SSH.

scp

Para transferir un archivo desde la maquina local a una maquina remota, se debe escribir

```
scp archivolocal usuario@maquina_remota:archivoremoto
```

Nótense los dos puntos ":" entre el nombre de la maquina remota y el nombre del archivo remoto.

Si no se indica el nombre de usuario (en este caso se debe suprimir también la "@"), se usa el nombre de usuario local.

Si no se indica el nombre de archivo remoto, se copia con el mismo nombre que tiene localmente, en el directorio home (remoto) del usuario.

El nombre del archivo remoto puede ser precedido por una ruta de directorio.

Ejemplos:

```
scp notas.txt eva:
```

Copia el archivo notas.txt a la maquina eva, usando el mismo nombre de usuario que estoy usando en la maquina local. El archivo es almacenado como notas.txt en mi directorio home de eva.

```
scp eva:notas.txt .
```

Copia el archivo notas.txt de mi home de eva al directorio actual ("." final indica directorio actual).

```
scp notas.txt otrousuario@eva:/tmp
```

Copia el archivo notas.txt a eva, esta vez usando un nombre de usuario distinto al que estoy usando en la maquina local. el archivo se almacena en el directorio /tmp

```
scp eva:mail/sent-mail-dec-2012 correos-viejos/enviados-dic-2003
```

Copia la casilla sent-mail-dec-2012 a la carpeta local correos-viejos, cambiándole el nombre.

Nota: El archivo de este ejemplo es el 'folder' del pine correspondiente a los mensajes enviados en ese mes.

FUENTES:

http://www.linuxtotal.com.mx/index.php?cont=info_admon_013
http://www.ubuntizandoelplaneta.com/2010/02/actualizando-debianubuntu-sin-saturar.html
http://eithel-inside.blogspot.com/2010/11/crear-un-mirror-de-los-repositorios-de.html

SCRIPT PARA BASH

Un **Script** (o guion) para Bash es un archivo de texto que contiene una sucesión de comandos de Shell que pueden ejecutar diversas tareas de acuerdo al contenido del texto del guión. De esta forma pueden automatizarse muchas acciones para alguna necesidad particular o para la administración de sistemas. El guión debe escribirse en un orden lógico pues Bash ejecutará el guión en el orden en que se escriben las lineas, de la misma forma que cuando se realiza una tarea cualquiera por una persona, por ejemplo; *primero hay que poner la escalera y luego subirse*.

Los Scripts de Bash deben tener siempre como primera linea del guión el texto, para que el Sistema Operativo ejecute la acción usando el programa Bash.

```
#!/bin/bash
```

Una vez escrito el Script y guardado en el disco en alguno de los directorios "bin" con el nombre y permiso de ejecución apropiados, se invoca, escribiendo en la consola el nombre del guión. Si el guión tiene una interfaz gráfica se invoca como otro programa cualquiera, uno o dos clic sobre el guión o su icono. Este puede escribirse en cualquiera de los editores de texto de Linux, por ejemplo pico ó vi y será ya un guión funcional cuando se salve a alguno de los "bin".

Es buena práctica cuando se escribe un guión salvarlo apenas se hallan escrito las primeras línea para ir comprobando su funcionamiento e ir corrigiendo los problemas.

VARIABLES

Es impensable elaborar Scripts de Bash sin el uso de las variables. Una variable es una estructura de texto (una letra, un número o sucesiones de ellos) que representa alguno de los elementos que varían en valor y/o significado en el entorno de la Shell, sirviendo como elemento básico de entrada/salida de valores a y desde los comandos en su ejecución consecutiva. Para invocar una variable se utiliza el carácter especial **$** precediendo al nombre de la variable.
Existen dos tipos de variables:

VARIABLES INTRÍNSECAS DE BASH.

Estas son elaboradas por defecto por el propio Bash y son:

- **$0** -> Nombre del guión
- **$1....$n** -> Variables que almacenan los n argumentos (opciones) proporcionados al comando.
- **$#** -> Variable que contiene el total de los argumentos proporcionados.
- **$*** -> Conjunto de los argumentos.
- **$?** -> Valor de ejecución del comando anterior, si es cero es que el comando anterior se ejecutó sin errores, de lo contrario hubo algún error.
- **$$** -> Identifica el proceso del guión.
- **$!** -> Identifica el último proceso arrancado en el trasfondo (background).

Ejemplo : Creamos este Script el cual llamaremos variables_int.sh

```
#!/bin/bash
echo "Fui creado con : $# parámetros"
echo "Mi nombre es : $0"
echo "Mi primer parámetro es : $1"
echo "Mi segundo parámetro es : $2"
echo "Todos los parámetros son : $@"
echo "El nombre del script es : $0"
echo "El PID del script es : $$"
```

Ahora para ejecutarlo, hay que de darle permiso de ejecución, y debemos de pasarle los tres parámetros que esta esperando, de la siguiente manera:

```
chmod +x variables_int.sh
./variables_int.sh primero segundo tercero
Fui creado con : 3 parámetros
Mi nombre es : ./variables_int.sh
Mi primer parámetro es : primero
Mi segundo parámetro es : segundo
Todos los parámetros son : primero segundo tercero
El nombre del script es : ./variables_int.sh
El PID del script es : 9186
```

VARIABLES CREADAS POR EL PROGRAMADOR.

Las variables pueden ser creadas en cualquier momento, pero siempre antes de su utilización de manera muy simple, se escribe:

```
nombre_variable=valor_variable
```

En cualquier momento posterior a la creación si se coloca $nombre_variable dentro del entorno de la Shell el sistema colocará allí valor_variable.

```
SALUDO=Bienvenido
```

En cualquier momento posterior si se pone $SALUDO, Bash colocará ahí Bienvenido.

Una variable también puede ser la salida de un comando si ponemos al principio y final del mismo un acento invertido.

```
SALIDA=`comando`
```

Le indicará al sistema que donde se escriba $SALIDA debe poner la salida de ese comando. Es práctica común utilizar mayúsculas para las variables a fin de

identificarlas fácilmente dentro del guión.

Cuando se ejecutan Scripts que pueden ser "hijos" de otro guión en ocasiones es necesario exportar las variables, esto se hace escribiendo:

export nombre_variable

CARACTERES ESPECIALES.

Existe un grupo de caracteres especiales (también llamados meta caracteres) que tienen significado propio para Bash. Algunos son:

- **** -> Le indica a Bash que ignore el carácter especial que viene después.
- " " -> Cuando se encierra entre comillas dobles un texto o una variables si esta es una frase (cadena de palabras) Bash lo interpretará como una cadena única.
- **$** -> Identifica que lo que le sigue es una variable.
- ' ' -> Las comillas simples se usan para desactivar todos los caracteres especiales encerrados dentro de ellas, así tenemos que si escribe '$VARIABLE' Bash interpreta literalmente lo escrito y no como variable.
- **#** -> Cuando se coloca este carácter dentro de una linea del guión, Bash ignora el resto de la linea. Muy útil para hacer comentarios y anotaciones o para inhabilitar una linea de comandos al hacer pruebas.
- **;** -> Este carácter se usa para separar la ejecución de distintos comandos en una misma linea de comandos.
- `` -> Se utiliza como se explicó en el punto anterior, para convertir la salida de un comando en una variable. El comando en cuestión se ejecuta en una sub shell.

También están **|**, **()**, **!**, **>**, **<**, cuyo significado se verá mas adelante. El espacio es otro carácter especial y se interpreta por bash como el separador del nombre del programa y las opciones dentro de la linea de comandos, por esta razón es importante encerrar entre comillas dobles el texto o las propias variables cuando son una frase de varias palabras.

Otro carácter que debe evitarse en lo posible su uso es el guión **(-)** ya que para la mayoría de los programas se usa para indicarle al propio programa que lo que sigue es una de sus opciones, de manera tal por ejemplo, si usted crea un archivo con nombre –archivo (en caso que pueda) después será difícil borrarlo ya que rm (programa que borra) tratará el archivo como una de sus opciones (al "ver" el Script) y dará de error algo así, "Opción –archivo no se reconoce".

PALABRAS ESPECIALES.

Hay un grupo de palabras que tienen significado especial para bash y que siempre que se pueda deben evitarse cuando se escriben lineas de comandos para no crearle "confusiones" algunas son: exit, break, continue, true, false, return etc... cuyo significado es mas o menos así:

- **exit** : Se sale del guión
- **break** : Se manda explícitamente a salir de un ciclo
- **continue** : Se manda explícitamente a retornar en un ciclo
- **return** : Como exit pero solo se sale del comando u operación sin cerrar

el guión
- **true** : Indica que una condición es verdadera
- **false ;** Indica que una condición es falsa

ARGUMENTOS PROPIOS DE BASH.

Bash como programa tiene algunos argumentos útiles y propios que se usan con frecuencia en la elaboración de Scripts en los condicionales vinculados a la determinación de elementos sobre los archivos, variables, cadenas de palabras o cadenas de pruebas, los mas comunes son:

1. Argumentos de Archivos -----> Cierto si.... (salida 0)

-d : Archivo existe y es un directorio
-c : Archivo existe y es de caracteres
-e : Archivo existe
-h : Archivo existe y es un vínculo simbólico
-s : Archivo existe y no está vacío
-f : Archivo existe y es normal
-r : Tienes permiso de lectura del archivo
-w : Tienes permiso de escritura en el archivo
-x : Tienes permiso de ejecución del archivo
-O : Eres propietario del archivo
-G : Perteneces al grupo que tiene acceso al archivo
-n : Variable existe y no es nula
Archivo1 nt Archivo2 : Archivo1 es mas nuevo que Archivo2
Archivo1 -ot Archivo2 : Archivo1 es mas viejo que Archivo2

2. Agumentos de cadenas -----> Cierto si

-z : La cadena está vacía
-n : La cadena no está vacía
cadena1 = cadena2 : Si las cadenas son iguales
cadena1 != cadena2 : Si las cadenas son diferentes

cadena1 <> Si la cadena 1 va antes en el orden lexicográfico
cadena1 >cadena2 : Si la cadena 1 va después en el orden lexicográfico

Ejemplo : Con la instrucción *test*, realizaremos los siguientes ejemplos:

```
test -r variables_int.sh
```

```
echo $?
0
```

Ejemplo : Ahora vamos a comprobar el resultado, cuando buscamos otro archivo, en este caso, buscaremos *bariable_int.sh*

```
test -r bariables_int.sh
echo $?
1
```

Tal como lo vimos, nos devuelve *$?* un *0* cuando la consulta da verdadero y *1* cuando la consulta da falso.

ENTRADA / SALIDA.

En algunas ocasiones será necesario leer ciertas variables desde el teclado o imprimirlas a la pantalla, para imprimir a la pantalla se pueden invocar dos programas en la linea de comandos:

- *echo*
- *printf* (que es un echo mejorado)

Y para leer desde el teclado se usa:

- *read*

Si hacemos un read sin asignar variable, el dato de almacena en *$REPLY* una variable del sistema. Tanto el comando *echo* como *read* tienen sus propias opciones.

Ejemplos:
1. Si creamos en una linea del Script una variable como un comando y queremos imprimir la variable a la pantalla podemos hacer algo así:

```
VARIABLE=`comando`
echo "$VARIABLE"
```

La palabra $VARIABLE está puesta entre comillas dobles para que se imprima todo el texto ignorando los espacios entre palabras.

2. Si escribimos en una linea del guión

```
read PREGUNTA
```

habremos creado una variable de nombre PREGUNTA así es que si luego ponemos

```
echo "$PREGUNTA"
```

Se imprimirá a la pantalla lo que se escribió en el teclado al presionar la tecla Enter.

Con los elementos tratados hasta aquí ya podemos escribir nuestros primeros Scripts

Script 1

```
#!/bin/bash
echo Hola mundo
```

Cuando se corre este guión se imprimirá a la pantalla Hola mundo

Script 2 -> Lo mismo usando una variable

```
#!/bin/bash
VARIABLE=Hola mundo
echo "$VARIABLE"
```

Nótese la variable entre comillas dobles para que imprima todo el texto.

Script 3 -> Cuando se usan mas de una variable

```
#!/bin/bash
VARIABLE=Hola
SALUDO=mundo
echo "$VARIABLE""$SALUDO"
```

En los tres casos se imprimirá a la pantalla Hola mundo

Script 4 -> Si se usan caracteres especiales la cosa puede cambiar

```
#!/bin/bash
VAR=auto
echo "Me compré un $VAR" Imprimirá Me compré un auto
echo 'Me compré un $VAR' Imprimirá Me compré un $VAR
echo "Me compré un \$VAR" Imprimirá Me compré un $VAR
```

Note como las comillas simples y el carácter \ hacen que Bash ignore la función del carácter especial $. Siempre las comillas simples harán que se ignore todos los meta caracteres encerrados entre ellas y \ solo el que sigue después.

CONDICIONALES.

Los condicionales son claves para "explicarle" a la máquina como debe proceder en una tarea cualquiera, esto se hace casi como si se estuviera explicando una tarea a ejecutar a otra persona.

- **if then fi**

  El condicional por excelencia tiene seis palabras claves que son if, elif, else, then y fi. Donde las palabras tienen un significado comunicativo (en

Inglés) casi literal, tal y cual se tratara con otra persona y que Bash por defecto las entienda con ese significado.

- **if -> si** condicional (de si esto o lo otro)
- **elif -> también si** (contracción de else if)
- **else -> De cualquier otra manera**
- **then -> Entonces**
- **fi -> if invertido**, indica que se acabó la condicional abierta con if

Solo son imprescindibles en la estructura del Script *if, then* y *fi*. Supongamos ahora que es usted el jefe de una oficina y tiene una secretaria y que por alguna razón le han pedido que envíe una copia de cualquier documento que lo identifique; normalmente le diría a la secretaria algo así:

"Maria, por favor, busca en el archivo alguna identificación" (condición a evaluar)

if "si es una copia del pasaporte" (primer resultado de la condición); then (entonces)" envíala por fax a...." (equivalente al comando a ejecutar) elif "si es de la licencia de conducción" (segundo resultado de la condición); then

"envíala por correo" (otro comando a ejecutar)

elif " si es del carnet de identidad" (tercer resultado de la condición);

then

"envíala con un mensajero " (otro comando diferente)

else "de cualquier otra manera"

"pasa un fax diciendo que la enviaré mañana" (otro comando)

fi

Observe que la acción a ejecutar (equivalente al comando) se hace si la condición se evalúa como verdadera de lo contrario se ignora y se pasa a la próxima, si ninguna es verdadera se ejecuta finalmente la acción después del else. La sintaxis de bash se debe tener en cuenta a la hora de escribir el Script o de lo contrario Bash no entenderá lo que usted quiso decirle, Pongamos ejemplos de guiones reales

Script 5

```
#!/bin/bash
VAR1=Pablo
VAR2=Pedro
if [ "$VAR1" = "$VAR2" ]; then
echo Son iguales
else
echo Son diferentes
fi
```

Los corchetes son parte de la sintaxis de Bash y en realidad son un atajo (shortcut) al programa test que es el que ejecuta la acción de comparación. Observe siempre los espacios vacíos entre los elementos que conforman la linea de comandos (excepto entre el último corchete y el ;), recuerde que ese espacio vacío por defecto Bash lo interpreta como final de un elemento y comienzo de otro. Si corre

este guión siempre se imprimirá a pantalla Son diferentes, ya que la condición es falsa. Pero si cambia el valor de VAR2=Pablo entonces se imprime Son iguales. Guión 6 Un guión que verifica si existe un directorio y si no existe lo crea e imprime mensajes a pantalla comunicando la acción ejecutada.

```
#!/bin/bash
DIR=~/fotos (crea como variable el directorio /home/fotos)
if [ ! -d "$DIR" ]; then (verifica si no existe el directorio)
mkdir "$DIR" (si la condición es cierta, no existe el directorio, lo crea)
if [ $? -eq 0 ]; then (verifica si la acción se ejecutó sin errores, de serlo imprime lo que sigue)
echo "$DIR" ha sido creado..."
else (de lo contrario imprime)
echo "Se produce un error al crear "$DIR"
fi (Se cierra la condición abierta en la realización del directorio segundo if)
else ( de lo contrario, relativo al primer if)
echo "Se usará "$DIR" existente"
fi
```

En este guión pueden verse varias cosas nuevas:

1. El carácter ! niega la acción, si se hubiera escrito if [-d "$DIR"] lo que se estaba evaluando era la condición ¿existe el directorio"$DIR"? pero al colocar ! se evalúa lo contrario.

2. El carácter ~ significa el /home del usuario.

3. La expresión -eq se utiliza cuando quieren compararse valores numéricos, y significa =

4. Se usa una de las variables del sistema "$?" explicada mas arriba.

5. Pueden utilizarse unos condicionales dentro de otros siempre que se cierren apropiadamente.

```
#!/bin/bash
echo "Diga si o no:"
read VAR
if [ "$VAR" = si ]; then
echo "Escribiste -si-"
elif [ "$VAR" = no ]; then
echo "Escribiste -no-"
elif [ "$VAR" = "" ]; then
echo "No puede dejarlo en blanco"
else
echo "Lo que escribió no se acepta"
fi
```

Observe que se está evaluando varias opciones de la misma condición por lo que lo apropiado es incorporar los respectivos elif dentro de la misma condicional. Un elemento nuevo que se incorpora aquí es la condición " " que quiere decir "la variable está vacía", en este caso, cuando no se escribió nada.

- **case-in esac**

Cuando una variable puede puede adquirir varios valores o significados diferentes, ya hemos visto como puede usarse la palabra elif para hacer diferentes ejecuciones de comandos dentro de una misma condicional if-then-fi de acuerdo al valor de la variable. Una forma de realizar la misma acción sin escribir tantas lineas de condicionales elif y con ello disminuir el tamaño del guión es la utilización de la sentencia case-in-esac. Esta sentencia permite vincular patrones de texto con conjuntos de comandos; cuando la variable de la sentencia coincide con alguno de los patrones, se ejecuta el conjunto de comandos asociados. La sintaxis de la sentencia case-in esac es como sigue

```
case "nombre_variable" in
posibilidad 1) "uno o mas comandos" ;;
posibilidad 2) "uno o mas comandos" ;;
posibilidad n) "uno o mas comandos" ;;
esac
```

Script 8

```
#!/bin/bash
echo "Diga si o no:"

read VAR
case "$VAR"
in
si) echo "Escribiste -si-" ;;
no) echo "Escribiste -no-" ;;
*) echo "Lo que escribió no se acepta" ;;
esac
```

Este Script es el mismo que el Script 7 pero utilizando la sentencia case-in-esac Observe que el carácter (*) utilizado en la última opción significa "patrón no contemplado" en este caso.

FUNCIONES

Como mecanismo de estructuración en la codificación de Scripts, existe la posibilidad de crear funciones. Su definición exige la definición de un nombre y un cuerpo. El nombre que debe ser representativo , es seguido de apertura y cierre de paréntesis, mientras que el cuerpo se delimita con llaves. La sintaxis es la siguiente.

```
nombre_función ()
{
```

```
uno o mas comandos
}
```

Una vez definida la función se utiliza como si de un comando se tratase, invocándolo con el nombre de la función. Hay que hacer una invocación de la función ya definida para que se ejecute el código en su interior y se convierta en operativa. Las funciones son muy útiles cuando segmentos del código del Script son repetitivos, de tal forma solo se escriben una vez y se invocan todas las veces que haga falta, practicando el divino arte de la recursión.

Creando una función simple:

```
ayuda () (se define la función ayuda)
{
echo "Las opciones son si o no, luego apriete Enter"
}
```

Después de creada y activada la función, cada vez que necesitemos la "ayuda" dentro del guión solo colocamos la palabra ayuda como si se tratase de un comando mas y Bash ejecutará el código incluido dentro de la función, es decir imprimirá el texto "Las opciones son si o no, luego apriete Enter". Las funciones pueden ser definidas en cualquier orden, pueden ser tantas como haga falta y pueden contener un paquete relativamente complejo de comandos. Un programador que ha pensado la estructura del Script antes de empezarlo puede y de hecho se hace, crear todas las funciones que necesitará al empezar el guión. Pruebe lo siguiente

Script 9

```
#!/bin/bash
salir () #(Se crea la función salir)
{
exit #(comando)
}
hola() #(Se crea la función Hola)
{
echo Hola #(comando)
}
hola # (Se invoca la función Hola)
salir # ( Se invoca la función salir)
echo "Esto no se imprime nunca"
```

Verá que el último echo no se imprime ya que primero se invoca la función hola y luego la función salir que cierra el guión (exit). Trate ahora poniendo un comentario (#) a la linea que invoca la función salir (linea 11) y note la diferencia, vera como se imprime el último echo. Observe también como se han comentado aquellas cosas que no son parte integrante del guión pero que se pueden escribir para hacer aclaraciones o anotaciones de interés.

CICLOS, LAZOS O BUCLES

- **While-do done**

La sentencia *while–do done* se utiliza para ejecutar un grupo de comandos en forma repetida mientras una condición sea verdadera. Su sintaxis es:

```
while

lista de comandos 1
do
lista de comandos 2
```

Mientras la condición de control (lista de comandos1) sea verdadera, se ejecutaran los comandos comprendidos entre *do* y *done* en forma repetida, si la condición da falsa (o encuentra una interrupción explícita dentro del código) el programa sale del bucle (se para) y continua la ejecución por debajo del *while*. Un ejemplo de la utilidad de este lazo es la posibilidad de poder escoger varias opciones de un menú sin tener que correr el guión para cada opción, es decir se escoge y evalua una opción y el programa no se cierra, vuelve al menú principal y se puede escoger otra opción, tantas veces como sea necesario. Veamos un ejemplo de como elaborar un menú de opciones.

```
#!/bin/bash
while [ "$OPCION" != 5 ]
do
echo "[1] Listar archivos"
echo "[2] Ver directorio de trabajo"
echo "[3] Crear directorio"
echo "[4] Crear usuario"
echo "[5] Salir"
read –p "Ingrese una opción: " OPCION
case $OPCION in
1) ls;;
2) pwd;;
3) read –p "Nombre del directorio: " DIRECTORIO
mkdir $DIRECTORIO;;
4) if id | grep uid=0
then
read –p "Nombre del usuario: " NOMBREUSUARIO
useradd $NOMBREUSUARIO
else
echo "Se necesitan permisos de root"
fi;;
5);;
*) echo "Opción ingresada invalida, intente de nuevo";;
esac
done
```

```
exit 0
```

Descripción del Script:

1. En la primera linea condicionamos el lazo a que la opción escogida sea diferente de 5.

2. Luego se hace una lista de echos de las opciones desde 1 hasta 5 con su descripción para que sean imprimidas a la pantalla y así poder escoger alguna.

3. Le sigue el comando *read* para que lea del teclado la opción escogida (variable OPCION), a *read* se le ha agregado –p que hace que imprima un mensaje, en este caso imprime Ingrese una opción.

4. Para ahorrar lineas del guión se elabora un case con los comandos que deben ejecutarse en cada caso ls para listar los archivos [1], pwd (present work directory) para ver directorio de trabajo [2], otro read para escribir el nombre del directorio que quiere crear [3] y hacer la variable DIRECTORIO seguido por *mkdir* que crea el directorio, luego se crea una condicional *if–fi* para chequear si el usuario tiene permisos de root, necesario para la opción [4] de crear un usuario rechazándolo de lo contrario, después viene la opción [5] vacía que ejecuta el comando *exit* 0, finalmente se incluye "cualquier otra cosa" con el carácter *

Este guión resulta interesante porque se usan las dos formas de compactar el guión vistas hasta ahora, la sentencia case–in esac y la while–do done. Además empiezan a aparecer incluidos en los comandos algunos de los programas muy usados de Linux al escribir guiones.

- **until–do done**

La sentencia *until–do* done es lo contrario de *while–do* done es decir el lazo se cierra o para, cuando la condición sea falsa. Si le parece que ambas son muy parecidas está en lo cierto. En ambos casos se pueden elaborar bucles o ciclos infinitos si la condición de control es siempre verdadera o falsa según el caso, veamos Lazos infinitos Bucles infinitos son aquellos donde la ejecución continua dentro del bucle indefinidamente, veamos como hacer un bucle infinito mediante *while*:

```
while true
do
comando 1
comando 2
comando n
done
```

La condición siempre es verdadera y se ejecutara el bucle indefinidamente, mediante *until* sería así:

```
until false
do
comando 1
comando 2
comando n
done
```

Existe la posibilidad de salir de un bucle, independientemente del estado de la condición, el comando *break* produce el abandono del bucle inmediatamente. Veamos el guión anterior sobre la creación de un menú utilizando un lazo infinito y el

comando *break*

```
while true
do
echo "[1] Listar archivos"
echo "[2] Ver directorio de trabajo"
echo "[3] Crear directorio"
echo "[4] Crear usuario"
echo "[5] Salir"
read -p "Ingrese una opción: " OPCION
case $OPCION in
1) ls;;
2) pwd;;
3) read -p "Nombre del directorio: " DIRECTORIO
mkdir $DIRECTORIO;;
4) if id | grep uid=0
then
read -p "Nombre del usuario: " NOMBREUSUARIO
useradd $NOMBREUSUARIO
else
echo "Se necesitan permisos de root"
fi;;
5)
echo "Abandonando el programa..."
break;;
*)
echo "Opción ingresada invalida, intente de nuevo";;
esac
doGuión 11ne
exit 0
```

- **for-in-done**

Es otro tipo de ciclo o lazo disponible, la diferencia con los anteriores es que no se basa en una condición, sino que ejecuta el bucle una cantidad determinada de veces, su sintaxis es la siguiente:

```
for variable in arg 1 arg 2 ......arg n
do
comando 1
comando 2
comando n
```

```
done
```

Ejemplos

```
for LETRA in a b c d e f
do
echo $LETRA
done
```

En este guión el comando echo se ejecutara tantas veces como argumentos se hayan puesto después del in, por lo tanto imprimirá seis lineas cada una con una letra de la a a la f.

```
for ARCHIVO in *
if [ -d $ARCHIVO ]; then
cd $ARCHIVO
rm *.tmp
cd ..
fi
done
```

Este es un guión entra en todos los subdirectorios del directorio actual de trabajo y borrará todos los archivos .tmp (temporales). En este caso el carácter * se usa en la primera linea con el significado "tantas veces como sea necesario" y en la penúltima linea como "cualquier cosa".

REDIRECCIONAMIENTO.

Es frecuente la necesidad de redirigir resultados de la ejecución de un comando a diferentes lugares, que pueden ser los descriptores de ficheros **stdin**, **stdout** y **stderr**, a la entrada de otro comando o a un archivo en el disco duro, esto se llama redirección y es muy útil en la escritura de guiones.

1. Los descriptores de archivos

En Bash al igual que en cualquier otro programa de consola de Linux tenemos tres flujos o descriptores de archivos abiertos por defecto:

- La entrada estándar **(STDIN)**
- La salida estándar **(STDOUT)**
- El error estándar **(STDERR)**

El primero puede ser utilizado para leer de él, y los otros dos para enviar datos hacia ellos. Normalmente **STDIN** viene del teclado de la terminal en uso, y tanto **STDOUT** como **STDERR** van hacia la pantalla. **STDOUT** muestra los datos normales o esperados durante la ejecución, y **STDERR** se utiliza para enviar datos de depuración o errores. Cualquier programa iniciado desde el shell, a menos que se le indique explícitamente, hereda estos tres descriptores de archivo permitiéndole interactuar con el usuario.

- Enviar STDOUT a un archivo

En ocasiones necesitamos enviar la salida estándar a un archivo y no a la pantalla, ya sea porque es muy grande para "manejar a ojo" o porque nos interesa guardarla a disco duro. Para enviar la salida estándar a un archivo usamos > con lo que se sobreescribe el archivo si ya existe, o >> que solo agrega los datos de salida al final del archivo ya existente.

```
#!/bin/bash
ls -R /home/mis_fotos > /tmp/indice
```

Creará un archivo llamado /tmp/indice donde estará el listado de los archivos bajo /home/mis_fotos.

- Tomar STDIN de un archivo

Si queremos que un proceso tome su entrada estándar de un archivo existente usamos <>

- Enviar STDERR a un archivo

Si queremos enviar la salida de errores a un archivo se procede igual que lo que se mencionaba con respecto a la salida estándar pero se usa &> o &>> según el caso.

- Enviar STDERR a STDOUT

Para esto se escribe al final de la linea de comandos 2>&1.

- Enviar STDOUT a STDERR

En este caso se escribe al final de la linea de comandos 1>&2

2.Entubado

Las tuberías se utilizan para enviar la salida de un comando o proceso a la entrada de otro, esto es con frecuencia necesario para completar una acción iniciada con un comando que debe ser completada con otro. Es simple el modo de operar, solo se coloca el carácter | en la linea de comandos entre un programa y otro. Este carácter (|) se conoce como tubo (pipe)

```
#!/bin/bash
file -b "$1" | grep -i "vorbis" >/dev/null 2>&1
if [ $? -eq 0 ]; then
oggdec "$1"
echo "Hecho"
else
echo "Archivo no soportado"
exit
fi
```

Este guión convierte a wav cualquier archivo de audio ogg. Primero se invoca a file para que analice el tipo de archivo correspondiente a la variable $1 que como ya se sabe es el primer argumento introducido en la linea de comandos (por ejemplo la ruta hasta un archivo). Luego la salida de file se entuba al programa grep que determina si dentro del archivo aparece la palabra vorbis (caso de los archivos de audio ogg).

El condicional *if- then-fi* chequea que sea cierto (es decir la palabra vorbis si existía, por lo que es un archivo ogg de audio), entonces se decodifica a wav con el comando oggdec, de lo contrario se imprime que es un archivo no soportado. Tanto la salida estándar como la de errores se envía a /dev/null, un dispositivo que "desaparece" la información suprimiendo la salida por pantalla. Esto es conveniente y saludable en muchas lineas de comandos cuando la salida puede generar gran cantidad de información tanto de salida estándar como de errores y estos no nos interesan. Solo se escribe >/dev/null 2>&1.

GLOBALES Y EXPANSIONES.

1. Globales

Estos son aliados cuando uno quiere ahorrarse teclazos y funcionan como "generalizadores" de cosas, los globales mas comunes son:

1. ~ Le dice a Bash que es el directorio home del usuario.
2. * Significa "todo lo que puedas incluir ahí" de forma tal que si ponemos el comando ls ~/*.wav listará todos los archivos .wav que están en el directorio home del usuario. Ahora si escribimos ls ~/m* nos listará todos los archivos de home que empiecen con m.
3. . Un punto en el entorno de la shell significa "el directorio donde estamos trabajando"

Ejemplo:

```
#!/bin/bash
DIR=.
mkdir "$DIR"
echo "$?"
```

Si escribimos este guión y lo corremos dará un error. Por supuesto, le estamos mandando a hacer el directorio donde estamos. Habrá notado usted que es muy común a la hora de compilar programas desde el binario utilizar ./configure, con esto le estamos diciendo a Bash "corre el archivo configure que está en este mismo directorio".

2. Expansiones.

Las expansiones son mas configurables y trabajan con argumentos mucho mas definidos, está claramente hecha para hacer mas inteligente la shell. Cuando especificamos una lista de valores o argumentos separados por comas entre llaves, Bash la expande convirtiéndola en la cadena expandida con cada uno de los argumentos, por ejemplo:

el comando

echo este/directorio/{algo,muy,demasiado}/largo

dará como resultado la impresión a pantalla de:

este/directorio/algo/largo este/directorio/muy/largo

este/directorio/demasiado/largo

Hay que tener en cuenta que:

a) La expansión funciona sobre una sola palabra sin espacios si escribimos:

echo esto {es,parece} difícil
escribirá:
esto es parece difícil

b) La expansión no se realiza entre comillas simples ni dobles por lo que no sirve para corregir el ejemplo anterior:

echo "esto {es,parece} difícil"
dará:
esto {es,parece} difícil

c) Lo que debe hacerse es ignorar o escapar los espacios y escribir

echo esto\ {es,parece}\ confuso
así obtendremos lo que queríamos:
esto es difícil esto parece confuso.

Pueden ponerse múltiples expansiones en una sola linea y se obtendrán todas las combinaciones posibles.

echo {una,otra}\ combinación\ { bastante,muy}\ difícil.
Responde
una combinación bastante difícil. otra combinación bastante difícil. una combinación muy difícil. otra combinación muy difícil

ARITMÉTICA DE BASH.

Se pueden ejecutar en Bash las principales acciones aritméticas entre las variables utilizando los signos:

- **+** suma
- **–** resta
- ***** multiplicación
- **/** división

Las operaciones tienen su sintaxis que debe ser respetada para que Bash lo haga adecuadamente.

- Pruebe esto en la shell o la linea de comandos (consola).

  echo 1+1

La respuesta será 1+1 porque bash lo interpreta como caracteres simples, para que realice la operación de suma hay que escribir:

echo $((1+1)) o
echo $[1+1]

Bash no maneja números fraccionarios solo números enteros por lo tanto si usted escribe:

echo $[3/4] la respuesta será cero, sin embargo si escribe:
echo $[4/2] la respuesta será correcta 2

- También podrá utilizar a expr para las operaciones de la forma siguiente:

  expr argumento1 signo argumento2

pruebe en la consola

expr 2+2 la respuesta será 2+2 porque no hemos dejado espacios en blanco entre signos y argumentos o

expr 2 + 2 (con espacios en blanco) la respuesta será 4 o
expr 4 / 2 la respuesta será 2

Cuando se use es signo * para la multiplicación debe anteponerle una barra invertida para que Bash no lo interprete como un global, sería:

*expr 10 * 10 la respuesta será 100*

El programa expr da sus resultados directamente a la salida estándar pero tampoco maneja números fraccionarios. Hay que observar siempre un espacio entre los argumentos.

- Para operar con fraccionarios debe entubar la expresión al programa bc de la forma siguiente:

  echo operación | bc -l por ejemplo;
  echo 3/4 | bc -l

El resultado será 0.75 o

echo 2+2.5 | bc -l

Devolverá 4.5 En algunas distribuciones el programa bc no se instala por defecto. Hay otras expresiones que Bash interpreta aritméticamente;

n1 -lt n2 Menor que

n1 -le n2 Menor o igual que

n1 -eq n2 Igual que

n1 -ge n2 Mayor o igual que

n1 -gt n2 Mayor que

n1 -ne n2 Distinto que

Ejemplo : Comparemos los siguientes números 2 y 3

```
test 2 -lt 3
echo $?
0
```

Tal como lo vimos, nos devuelve *$?* un *0* cuando la consulta da verdadero y *1* cuando la consulta da falso.

LÓGICA DE BASH.

Para la shell los caracteres que tienen un significado lógico en la comparación o evaluación de archivos son:

> Mayor que

< Menor que

>= Mayor o igual que

<= Menor o igual que

! Diferente que

|| OR (ó)

&& AND (y)

```
#!/bin/bash
ARCHIVO=$1
file -b "$1" | grep -i 'JPEG' || file -b "$1" | grep -i 'GIF' || file -b "$1"
| grep -i 'PNG' || file -b "$1" | grep -i 'BITMAP' >/dev/null 2>&1
if [ $? -eq 0 ]; then
echo "Es una imagen"
else "No es una imágen"
fi
```

En este guión hemos supuesto que un archivo cualquiera se convierte en la variable $1 y queremos averiguar si el archivo es una imágen en alguno de los formatos mas comunes, primero acudimos a file para que "lea" el texto que contiene el archivo y lo entubamos a grep que buscará patrones de texto de lo que le entrega file. Como necesitamos averiguar si alguno de los patrones JPEG, GIF, PNG o BITMAP aparece dentro del archivo utilizamos varias instancias de file y grep separadas con OR (||), de esta forma le estamos diciendo en el comando "busca si aparece JPEG o GIF o PNG o BITMAP, si lo encuentras entonces imprime"

"Es una imágen" de cualquier otra forma imprime "No es una imágen"

LOS PROCESOS EN LINUX (NIVEL USUARIO)

¿ Que es un proceso ?

Un proceso simplemente es un programa en ejecución. Los procesos además de la información propia del programa contienen la información necesaria para que el programa interaccione con el sistema.

Tipos de procesos

- **Child (hijos)** : Un proceso hijo es un proceso creado por otro proceso, estos se crean mediante la llamada al sistema fork() y en realidad, todos los procesos en algún momento son hijos, todos menos el proceso init. En el caso de que un proceso sea creado mediante la shell (ejecutado desde esta), la shell sera el padre.
- **Orphan (huérfanos)** : Normalmente un proceso hijo termina antes que un proceso padre, pero se puede dar la situación de que se mate a un proceso padre (killed) y el hijo se quede sin padre (que crueldad). Entonces el proceso init lo adoptara como hijo, pero como su padre original no existe, es considerado huérfano.
- **Daemon (demonios)** : Es un tipo especial de proceso que se ejecuta en segundo plano y no esta asociado a ninguna shell. Esto se consigue matando la shell que crea el proceso, de esta forma el padre de este proceso pasa a ser el proceso init (queda huérfano). Estos corren con permisos de root y su cometido es proveer servicios a otros procesos.

- **Zombie** : Cuando un proceso hijo termina, el sistema guarda el PID (Identificador) y su estado (un parámetro) para dárselo a su padre. Hasta entonces el proceso finalizado entra en estado zombie. Cuando un proceso finaliza toda la memoria y recursos asociados con dicho proceso son liberados, pero la entrada del mismo en la tabla de procesos aún existe, para cuando su padre llame a la función wait() devolverle su PID y estado.

Administración de procesos

La administración de los procesos podemos hacerla mediante la interfaz gráfica o mediante la shell .

Buscando en los bosques y arboles

You can also list different information about each process. The --forest option makes it easy to see the process hierarchy, which will give you an indication of how the various processes on your system interrelate. When a process starts a new process, that new process is called a "child" process. In a --forest listing, parents appear on the left, and children appear as branches to the right:

```
$ ps x --forest
  PID TTY      STAT   TIME COMMAND
  927 pts/1    S      0:00 bash
 6690 pts/1    S      0:00  \_ bash
26909 pts/1    R      0:00      \_ ps x --forest
19930 pts/4    S      0:01 bash
25740 pts/4    S      0:04  \_ vi processes.txt
```

Using top

If you find yourself running ps several times in a row, trying to watch things change, what you probably want is top. top displays a continuously updated process listing, along with some useful summary information:

```
$ top
 10:02pm  up 19 days,  6:24,  8 users,  load average: 0.04, 0.05, 0.00
75 processes: 74 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  1.3% user,  2.5% system,  0.0% nice, 96.0% idle
Mem:   256020K av,  226580K used,   29440K free,       0K shrd,    3804K buff
Swap:  136544K av,   80256K used,   56288K free                  101760K
cached
```

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
  628 root      16   0  213M  31M  2304 S       0  1.9 12.5  91:43 X
26934 chouser   17   0  1272 1272  1076 R       0  1.1  0.4   0:00 top
  652 chouser    11    0 12016 8840   1604 S          0   0.5   3.4    3:52
gnome-termin
  641 chouser    9   0  2936 2808  1416 S       0  0.1  1.0   2:13 sawfish
```

Mediante la shell

Para la administración de procesos en la linea de comandos tenemos unas pocas instrucciones que nos van a ayudar con el cometido. Los comandos están dividido cuatro cometidos, visualización, terminación de procesos, cambio de prioridad y ejecución en segundo plano.

Para visualizar el estado del proceso seleccionado o de todos los procesos tenemos ps:

Comando ps

Otra alternativa para la visualización de los procesos es pstree, pstree nos muestra todos los procesos en forma de árbol.

Pero si lo que queremos es monitorizar los procesos en tiempo real disponemos del comando top, este muestra una lista de procesos que se actualiza cada 3 segundos (por defecto). Los procesos están ordenados por el uso de CPU y muestran PID, usuario, %CPU, %MEM.

Terminación de procesos

Kill: matar un proceso usando su PID

La forma más complicada pero al mismo tiempo más precisa de matar un proceso es a través de su PID (siglas en inglés de “Identificador de Proceso”). Cualquiera de estas 3 variantes puede servir:

kill -TERM pid

kill -SIGTERM pid

kill -15 pid

Se puede usar el nombre de la señal (TERM o SIGTERM) que se desea mandar al proceso o su número de identificación (9). Para ver un listado completo de las posibles señales, sugiero verificar el manual de kill. Para ello, ejecutá:

man kill

Veamos un ejemplo de cómo matar Firefox:

Primero, hay que averiguar el PID del programa:

ps -ef | grep firefox

Ese comando devolverá algo parecido a esto:

1986 ? Sl 7:22 /usr/lib/firefox-3.5.3/firefox

Usamos el PID devuelto por el comando anterior para aniquilar el proceso:

kill -9 1986

killall: matar un proceso usando su nombre

Este comando es bien fácil

killall nombre_proceso

Un dato a tener en cuenta al usar este método es que en caso de que haya más de una instancia de ese programa ejecutándose, se cerrarán todas.

Cambio de prioridad

La prioridad de proceso, se utiliza para decidir la cantidad de tiempo que el proceso podrá utilizar el procesador, por intervalo de tiempo. Paso a explicarlo, el/los procesadores son compartidos por varios procesos (los procesos van alternándose en el uso del o de los procesadores) dando la sensación al usuario que todas las aplicaciones, tareas, procesos se ejecutan a la vez, pues bien la prioridad le dice al sistema que procesos pueden utilizar mas tiempo de procesador y que procesos pasan a un segundo lugar. Esto puede llegar a ocasionar que la ejecución de algún/os proceso/s no llegue/n a ejecutarse nunca, ya que van siendo desplazados en la cola de procesos hacia el final por otros procesos con una prioridad mayor.

Mayor prioridad -20 (menos veinte)

Menor prioridad 19(diecinueve)

Si iniciamos un programa normalmente, y no hay ninguna configuración para el usuario o grupo que lo modifique, este se iniciará con prioridad 0 (cero)

nice

nice asigna una prioridad concreta a un programa al ser ejecutado, y por herencia las tareas y procesos que este programa pueda desencadenar.

sintaxis de nice

```
nice [argumento] [comando [argumentos-del-comando]]
```

Los argumentos se pueden pasar de 3 formas

- prioridad precedida de un - (a las prioridades positivas les da un aspecto negativo p.e. prioridad 12 -> nice -12 programa)
- prioridad detrás de -n (p.e. nice -n 12 programa)
- prioridad tras -adjustment= (p.e. nice -adjustment=12 programa)

Para lanzar un proceso ajustando la prioridad deberemos ejecutarlo con el nice y la opción -n con la variación de prioridad. La prioridad va desde el -20 (más favorable) a 19 (menos favorable). Así, por defecto, si lanzamos:

```
$ nice -n 10 comando
```

renice

renice utiliza los parámetros de la misma forma que nice

Consideraciones

- Cuando se inicia un programa con nice sin argumentos este comienza con una prioridad de 10.
- Tanto nice como renice nos permiten cambiar la prioridad de programas o procesos mediante la consola sin interferir en la ejecución del programa o proceso.
- Si queremos cambiar la prioridad a un proceso, deberemos utilizar el pid de dicho proceso.
- Podemos cambiar la prioridad de varios procesos a la vez p.e. renice prioridad pids -u usuarios
- Podemos utilizar y combinar cambios de prioridad para los procesos independientes con su pid, usuarios y grupos.
- Solo root puede utilizarlos para da incrementar la prioridad.
- Cualquier usuario puede utilizarlos para decrementar la prioridad a los procesos sobre los que tenga permiso

Por ejemplo, para augmentar la prioridad del proceso a -10 usaremos la opción -p indicando la nueva prioridad en lugar de la variación sobre la prioridad actual como en el caso del nice:

```
$ renice -10 -p 123
123: old priority 0, new priority -10
```

También podemos hacer el renice para todos los procesos de un usuario mediante la opción -u:

```
# renice -10 -u apache
513: old priority 0, new priority -10
```

Ejecutar procesos en segundo plano

Si queremos ejecutar un proceso en segundo plano (background) se utiliza el comando nohup o el operador &.

Si queremos que la terminal quede liberada de un proceso utilizamos el operador &, esto se utiliza mucho cuando ejecutamos un programa con interfaz gráfica mediante la linea de comandos. Si no utilizamos el operador & (ampersand) después de la llamada al programa la terminal queda inutilizada (esta ejecutando el programa), pero si utilizamos & el programa se ejecuta en segundo plano y podemos seguir llamando a mas programas.

Un ejemplo del ampersand seria:

```
gedit &
```

Al finalizar una sesión en un terminal se envía un signal (SIGHUP) a todos los procesos que esté ejecutando nuestro usuario. Como resultado, dichos procesos se mueren (aunque les hayamos puesto & al final). Para evitar esto utilizamos el comando nohup. Este comando hace que un proceso ignore la señal SIGHUP, y redirige la salida de nuestro programa a un archivo nohup.out que es creado en el directorio actual.

Un ejemplo del comando nohup junto al ampersand :

```
nohup ./script.sh &
```

donde script.sh es un Script en nuestra situación actual en el sistema de ficheros.

Free :

```
#free
             total       usado       libre     compart.     búffers     almac.
Mem:       3946592    3113176     833416           0      29864     744472
-/+ buffers/cache:    2338840    1607752
```

Intercambio: 999420 121440 877980

DPKG: TRABAJANDO CON PAQUETES .DEB

Recordamos algunas opciones de dpkg:

Para instalr un paquete deb usamos:

 # dpkg -i nombredelpaquete.deb

Para eliminar el paquete instalado, debemos poner:

 # dpkg -r nombredelpaquete

Tambien podemos usar para eliminar un paquete el parámetro --purge(-P)

 # dpkg -P nombredelpaquete

Con esto borramos la aplicación y los archivos de configuración.

Ahora si solo queremos ver el contenido del paquete deb podemos poner

 # dpkg -c nombredelpaquete.deb

Para obtener información acerca del paquete tal como el nombre del autor, el año en que fue compilado y una descripción corta de su uso podemos poner

 # dpkg -I nombredelpaquete.deb

Para conocer si tenemos instalado un determinado paquete podemos poner

 # dpkg -s nombredelpaquete

Si nosotros queremos conocer que archivos nos instala una determinada aplicación podemos poner

 # dpkg -L nombredelpaquete

Si queremos saber a qué paquete pertenece un fichero, podemos poner:

 # dpkg -S nombredefichero

dpkg-reconfigure: Reconfigurando los paquetes instalados

Cuando instalamos un paquete con APT, se descarga, se descomprime y por último se configura. Si queremos configurar de nuevo un paquete ya instalado usamos la instrucción dpkg-reconfigure.

Con la opción -p podemos indicar el nivel de detalle que se hará la configuración: low (bajo) o high (alto). Por ejemplo dpkg-reconfigure debconf

Servicios

Linux ofrece multitud de servicios o servidores, estos pueden iniciar o arrancar junto con la carga del sistema o pueden después ser puestos a funcionar cuando se requieran (es lo mejor). Parte esencial de la administración de sistemas Linux es continuamente trabajar con los servicios que este proporciona, cosa que es bastante sencilla.

Iniciando servicios manualmente, directorio init.d

Dentro de esta carpeta ubicada en /etc o en /etc/rc.d dependiendo de la distribución, se encuentran una serie de scripts que permiten iniciar/detener la gran mayoría de los servicios/servidores que estén instalados en el equipo. Estos scripts están programados de tal manera que la mayoría reconoce los siguientes argumentos:

- start
- stop
- restart
- status

Los argumentos son autodescriptivos, y tienen permisos de ejecución, entonces siendo root es posible iniciar un servicio de la siguiente manera, por ejemplo samba:

#> /etc/rc.d/init.d/smb start

Starting Samba SMB daemon [OK]

Solo que hay que cambiar start por stop | restart | status para detenerlo, reiniciarlo (releer archivos de configuración) o revisar su estatus. Ahora bien si estás parado dentro del directorio puedes hacerlo asi.

#> pwd

/etc/rc.d/init.d

#> ./smb stop

Shutting down Samba SMB daemon [OK]

Se trata de tan solo un script, así que con el permiso de ejecución (x) puedes ejecutarlo con ./ seguido del nombre del servicio, sin espacios y después el argumento que necesites, iniciarlo, detenerlo etc.

El comando service

En varias distribuciones, como Fedora o RedHat, existe el comando service, este comando permite también iniciar y/o detener servicios, de hecho funciona exactamente igual a como si escribiéramos la ruta completa hacía el directorio init.d, con service se indica de la siguiente manera:

#> service mysql status

Checking for service MySQL: stopped

Si se desea iniciarlo:

#> service mysql start

Starting service MySQL [OK]

Iniciando servicios desde el arranque del sistema

Parte de los servicios de Ubuntu se encuentran en /etc/init.d mientras que otros ya se encuentran migrados a Upstart, por este motivo se manipulan de diferentes formas.

Upstart tiene como objetivo reemplazar los daemons tradicionales de SystemV que gestionan las tareas a ejecutar en el arranque, la parada y puesta en marcha de servicios.

Upstart busca sustituir al daemon init, el primer proceso que se lanza en Linux tras cargar el kernel y que se encarga de arrancar el resto. init es el proceso padre de todos aquellos procesos que hayan perdido a su padre (es el padre de todos los daemons). El comando *pstree* permite ver esto gráficamente.

Por qué Upstart

init es un proceso síncrono que bloquea la ejecución de tareas hasta terminar con la actual. Las tareas que init debe ejecutar han de ser definidas con antelación y éstas sólo se ejecutan cuando init cambia su estado (generalmente porque la máquina se ha encendido, se está apagando o se está reiniciando). El daemon init decide qué tareas ejecutar al cambiar su estado (RUNLEVEL) mirando en el directorio /etc/rcX.d/, donde X indica el RUNLEVEL actual.

Este hecho impide que init gestione correctamente otras tareas que son necesarias ejecutar NO al cambiar de RUNLEVEL sino cuando se generan ciertos eventos, como por

ejemplo, las siguientes:

- Se quiere ejecutar un backup del servidor de la base de datos en cuanto se detecte que dicho servicio se ha parado
- Se conecta en caliente un dispositivo USB o disco externo
- Se quiere realizar un sondeo de los dispositivos de almacenamiento disponibles sin que se bloquee el sistema (especialmente cuando el disco a sondear está en estado stand-by y se enciende al detectar el sondeo)
- Se quiere ejecutar un script cada hora pero sólo si la ejecución anterior ya ha terminado

Hay más ejemplos y casos de uso en la discusión para reemplazar init en Ubuntu.

El modelo basado en eventos de Upstart permite responder de forma asíncrona a eventos como los mencionados, en cuanto éstos ocurren. La asincronía permite poder ejecutar en paralelo distintas tareas con el objetivo de minimizar el tiempo de arranque.

Iniciar y detener manualmente los servicios.

Este procedimiento permite iniciar y detener manualmente los servicios para un momento específico, es decir, no perdura en el tiempo, después de reiniciar el sistema operativo el servicio quedará nuevamente como haya sido configurado inicialmente.

Los servicios basados en /etc/init.d se manipulan de la siguiente manera.

```
$ sudo /etc/init.d/NOMBRE_SERVICIO stop      # detener

$ sudo /etc/init.d/NOMBRE_SERVICIO start     # iniciar
```

Los servicios basados en Upstart se manipulan de la siguiente manera.

```
$ sudo service NOMBRE_SERVICIO stop          # detener

$ sudo service NOMBRE_SERVICIO start         # iniciar
```

Activar y desactivar los servicios.

Este procedimiento permite determinar si se desea que un servicio se inicie o no automáticamente después de iniciado (boot) el sistema operativo.

Los servicios basados en /etc/init.d se manipulan de la siguiente manera.

```
$ sudo update-rc.d NOMBRE_SERVICIO disable   # desactivar

$ sudo update-rc.d NOMBRE_SERVICIO enable    # activar
```

Para desactivar los servicios basados en Upstart se debe editar el archivo /etc/init/NOMBRE_SERVICIO y comentar la linea que empieza con start on. Por ejemplo, para activar el servicio cron se debe realizar el siguiente procedimiento.

```
$ sudo vi /etc/init/cron.conf

# cron - regular background program processing daemon
#
# cron is a standard UNIX program that runs user-specified programs at
# periodic scheduled times

description    "regular background program processing daemon"

# start on runlevel [2345]
stop on runlevel [!2345]

expect fork
respawn

exec cron
```

Para activar nuevamente el servicio será necesario remover el comentario al comienzo de la línea start on.

Determinar el estado de un servicio.

Para determinar si un servicio se encuentra o no ejecutándose en un momento dado se debe realizar el siguiente procedimiento si el servicio se encuentra basado en /etc/init.d.

```
$ sudo /etc/init.d/NOMBRE_SERVICIO status
```

Si el servicio se encuentra basado en Upstart se debe hacer lo siguiente.

```
$ sudo status NOMBRE_SERVICIO
```

Mark Shuttleworth anunció que Upstart ha perdido la batalla en contra de systemd y en consecuencia Ubuntu adoptará este último. El anuncio del ex CEO de Canonical llega pocos días después de que la gente de Debian anunciara su decisión de

utilizar de ahora en adelante systemd.

La decisión de abandonar Upstart en favor de systemd facilitará el trabajo de los desarrolladores de Ubuntu, quienes tenían dos opciones: seguir los pasos de Debian o implementar los parches necesarios para poder seguir utilizando Upstart de manera eficiente.

FSTAB

Es necesario tener una pequeña definición de algunos conceptos antes de comenzar:

- *FileSystem: Todo medio físico que pueda almacenar archivos debe tener un filesystem para ser capaz de cumplir dicha función (ejemplo: una partición de un disco duro). Un filesystem es un sistema utilizado para organizar los archivos en dicho medio de almacenamiento, pero podríamos verlo como el propio medio de almacenamiento (a nivel de usuarios). Es necesario aclarar que esta no es una definición formal, pero nos aproximara al concepto…*
- *Tipo de filesystem: Como ya vimos un filesystem es un sistema de organización y es razonable que hayan varios sistemas distintos para organizar los archivos, cada uno con sus pro y sus contras. Por ejemplo: FAT, NTFS, EXT2, EXT3, EXT4, etc.*
- *Punto de Montaje: El punto de montaje es una carpeta o directorio. Luego de montarse el filesystem en dicho directorio podremos acceder a los archivos mediante él (directorio).*
- *Opciones de montaje: Permiten especificar ciertos parámetros para que al montarse el filesystem se haga de una forma especial, por ejemplo: ro (read-only) esto hace que no se puedan crear, modificar ni borrar archivos en ese filesystem. Otro ejemplo: errors=remount-ro (remount as read-only) en caso de algun error grave, el filesystem se monta en modo read-only.*
- *Dump: Dump es una herramienta de backups. Cuando el numero en esta columna es 0 (cero), dump ignorara ese filesystem.*
- *Pass: Comenzaremos explicando que es fschk. fschk es una herramienta para chequear los filesystems en busca de errores, etc. Cuando el numero en esta columna es 0 (cero), fschk ignorara ese filesystem.*

Trabajando con el archivo fstab

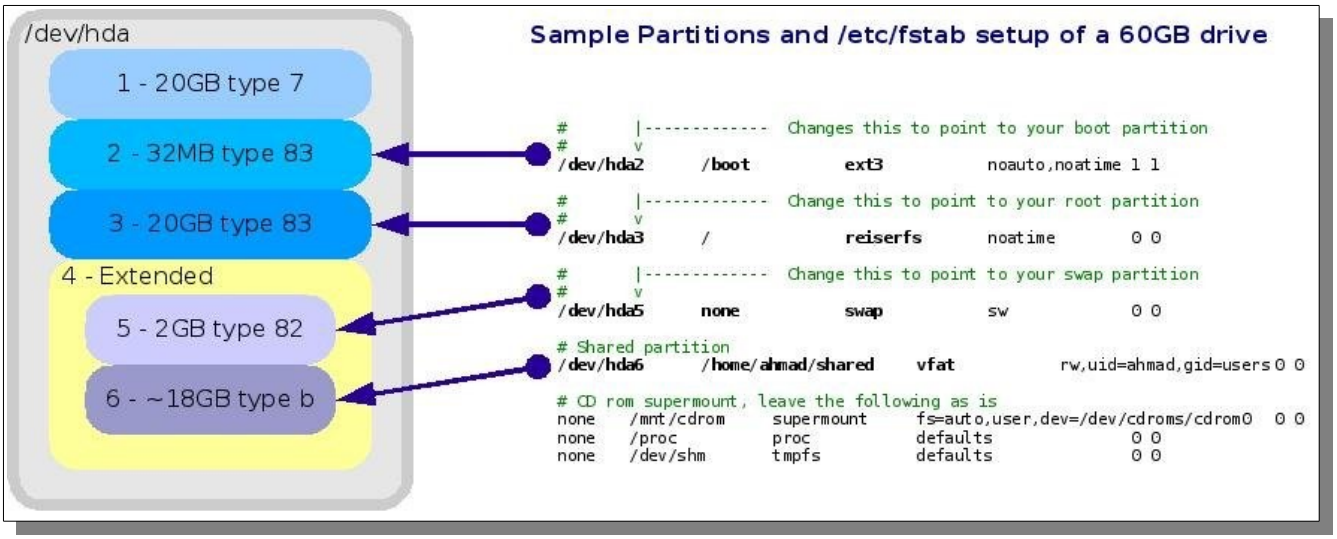
En primer lugar veremos la estructura de este archivo:

En este archivo cada una de las lineas hace referencia a un sistema de archivos (filesystem) y cada una de éstas respetan la siguiente estructura:

Veamos un ejemplo:

```
UUID=d4f1ec7e-f3d3-4bd4-becf-4f6da208237f  /  ext3  errors=remount-ro  0  1
/dev/sda5  /home  ext3  defaults  0  2
```

Ya notaron que en la primera linea se utiliza el UUID (Identificador Universal Único, por sus siglas en inglés) del filesystem y en la segunda la ruta del mismo (no me refiero al punto de montaje). Si utilizamos el UUID, nuestro método sera mucho mas robusto.



¿Cómo obtener la UUID correcta para cada partición?

Para esto deben ejecutar como root (o utilizando sudo como en el ejemplo) la siguiente línea:

```
sudo blkid
```

Y veremos algo como esto:

```
/dev/sda1: UUID="B6F0C97EF0C94579" TYPE="ntfs"
/dev/sda5: UUID="d4f1ec7e-f3d3-4bd4-becf-4f6da208237f" TYPE="ext3"
/dev/sda6: UUID="b8146e8f-77aa-44b8-9b37-5a2a90706eea" TYPE="ext3"
/dev/sda7: UUID="57cfda85-b5ce-4288-b42e-c19dc57a65d9" TYPE="swap"/dev/sdb1:
LABEL="Backup" UUID="5D9A907246C7446B" TYPE="ntfs"
```