Laboratorio Sesión 02: Más sobre gestión de usuarios y programación

En esta práctica veremos como gestionar las cuotas de disco de los usuarios y más opciones para la programación de scripts en `bash`.

2.1. Cuotas de disco.

Vamos a ver como gestionar las cuotas de disco desde Linux. Para ello, y debido a la instalación .ªespecial"que tenemos en laboratorio, en primer lugar deberemos incluir nuevos paquetes en el sistema sin actualizarlo.

Cambiar el sistema incluyendo nuevos paquetes es muy fácil (veremos más en próximas prácticas). Ahora vamos a salir del paso lo mejor que podamos :-). Seguid las siguientes instrucciones al pie de la letra (si todo va bien, en un par de prácticas entenderéis exactamente lo que estáis haciendo). A partir de aquí acordaros de usar `sudo` siempre que sea necesario.

Bien, primero vamos a instalar el sistema de cuotas. Es tan simple como ejecutar las siguientes instrucciones:

```
apt-get update
apt-get install quota
apt-get install quotatool
```

Ahora ya tenéis el sistema de cuotas instalado, vamos a continuar por el último apartado de la práctica anterior (con ligeros cambios, ya veréis):

En Linux el sistema básico de cuotas se gestiona a través de cuotas de usuario y de cuotas de grupos, aunque la tendencia actual es ignorar estas últimas. En primer lugar hay que especificarle al sistema que sistemas de ficheros queremos que tengan cuotas. Para ello deberéis editar el fichero "fstab" y añadir en las opciones del sistema de ficheros raíz (ya que es el único que tenemos, sería ideal usar cuotas solo para la partición con los datos de los usuarios) la opción "usrquota" o la opción "grpquota" (si quisiéramos cuotas de grupos), o la opción "quota" que hace lo mismo que "usrquota". En este caso añadid "quota". A continuación deberemos remontar el sistema para que los cambios tengan efecto:

```
mount / -o remount
```

Con esta opción se activan los comandos de inicio del sistema de cuotas. A continuación deberíais reiniciar el sistema para que las cuotas se activaran. Sin embargo, para evitar problemas al rearrancar el ordenador podemos probar a hacerlo a mano tecleando la siguiente orden:

```
quotacheck -vugm -a
```

En realidad este sistema no es el ideal, ya que activamos las cuotas con el sistema encendido (lo cual no es bueno).

Pregunta 2.1: *¿Para qué sirve la opción* `-m`*? Lógicamente esta opción no debe de estar incluida en la orden que se debe de ejecutar cada vez que el sistema arranque.*

Pregunta 2.2: *¿Qué fichero/s ha creado la orden* `quotacheck` *en el directorio raíz? Fijaros que esto solo lo hará la primera vez que se ejecute.*

Pregunta 2.3: *Fijaros que el sistema de quotas también os ha sugerido que uséis cuotas* `journaled`*. ¿Qué opciones en el fichero* `fstab` *deberíais poner en vez de* `quota` *para usar este tipo de cuotas? ¿Qué significa cada una?*

A continuación tenéis que activar el sistema de cuotas con la orden:

```
quotaon -avug
```

Recordad que ambos comandos se ejecutan de forma automática al reiniciarse el sistema con la opción correcta incluida en el sistema de ficheros (ya veremos como), así que lo lógico es no teclearlos en un sistema en producción.

A continuación podéis usar `edquota` para editar manualmente las cuotas, pero lo más cómodo suele ser usar el comando `quotatool` (que funciona en línea de comando). Vamos a verlo. Primero probad a ejecutar la opción `quota`, a secas, os dará la cuota actual del usuario que seáis (existe la opción `whoami` para saber que usuario sois). En un principio el sistema de cuotas está activado pero no hay cuotas asignadas. Podéis probar con la orden:

```
quotatool -u alumne -b -l 1000 /
```

para asignar una cuota al usuario alumne. A continuación cambiad a este usuario y ejecutad:

```
cd
echo "hola" > borrar.txt
```

Pregunta 2.4: *¿Que mensaje os da? Tenéis los comandos* `quota` *y* `du` *(ejecutado desde el "home") para comprobar que efectivamente habéis dejado poca cuota.*

Pregunta 2.5: *¿Cómo asignaríais una cuota de 16 Mbytes al usuario "alumne"? ¿Y una de 16 Mbytes pero que le damos 10 días con un margen de 1 Mbyte más?*

Pregunta 2.6: *¿Sabríais hacer un programa que dada una lista de usuarios en un fichero les pusiera a todos una cuota de 5 Mbytes? ¿Como os aseguraríais de que no le quitaseis a ninguno ficheros que ya tiene ni le rebajaseis la cuota si tiene más de dichos 5 Mbytes?*

Finamente, podéis probar otros comandos útiles sobre cuotas:

- `quotaoff`: para apagar el sistema de cuotas.
- `repquota`: información sobre las cuotas (para el root).
- `quotastats`: estadísticas sobre el sistema de cuotas (para desarrolladores principalmente).

2.2. Modificación del entorno de trabajo.

Ahora vamos a ver como podéis desde Linux modificar el entorno de trabajo. Lo primero que debéis saber es que en Linux, cuando trabajáis en línea de comandos, en realidad trabajáis con un programa "shell". Como con tantas otras cosas, podéis elegir con que programa "shell" trabajar.

Aquí vamos a ver el funcionamiento de la "shell" `bash`. Cada shell es independiente de las demás, pero la mayoría de características son comunes, aunque puede haber algunas diferencias. `bash` es quizás la mejor o la que más me gusta a mi :-) La shell más común era la Bourne (`sh`) y la mayoría de lo que voy a explicar se podría aplicar a esta directamente ya que `bash` es una evolución de `sh`. Otras shells interesantes (o bastante usadas) son la `csh`, la `tcsh` o la `zsh`.

Fijaros que como la shell es un programa que nos contacta con el núcleo, sus características solo dependen de la versión de la propia shell, no del núcleo ni de la distribución (que solo es un conjunto de programas) así que realmente esto que vamos a ver funciona en todos los sistemas Linux con `bash` (de hecho tb funciona en todos los sistemas Unix y Hurd...).

En primer lugar probad a cambiar de shell para ver que shells tenéis en el sistema. Podéis probarlas o escribiendo directamente el nombre, a ver si arrancan, o consultado el directorio `/bin` que es donde están los programas. La información sobre que shells admite el sistema está en el fichero `/etc/shells`, aunque no todas tienen porqué estar instaladas en `/bin`.

Pregunta 2.7: *¿Cuántas shells habéis encontrado? ¿Cuáles? ¿Qué sucede si instaláis la* `zshell` *(con* `sudo apt-get install zsh`*)?*

Ficheros asociados a la shell:

- `/bin/bash`: es el programa de la shell propiamente dicho. Fijaros que la shell por defecto que se os arrancará será la especificada en la última entrada del fichero `/etc/passwd` en la línea del usuario:

  ```
  root:x:0:0:root:/root:/bin/bash
  ```

  Si queréis arrancar una shell desde otra basta con que ejecutéis el comando correspondiente (ej. `csh` o `/bin/tcsh`, depende del PATH). Para salir de una shell basta con teclear exit. Ojo porque si es la consola principal en texto desconectaréis el Linux.

- `~/.bashrc`: (`~` es un atajo y se refiere al directorio "home" del usuario que se especifica en la penúltima entrada de la línea de usuario del fichero `/etc/passwd`). El fichero `.bashrc` lo que hace es guardar las ordenes que se ejecutan al principio cada vez que se invoca la shell interactiva `bash`. Es decir, es un fichero de órdenes de inicialización. Hay uno en cada directorio de usuario de forma que se puede personalizar (habitualmente, se copia de un esqueleto estándar que se encuentra en `/etc/skel/.bashrc`). Además es común que el fichero local `.bashrc` use entradas de un fichero común (a veces se llama `/etc/bashrc`, a veces `/etc/bash.bashrc`). Ojo con la diferencia, del primero se hace una copia local a cada usuario mientras que el segundo se referencia de forma que en el momento en que se modifique se modifica para todos los usuarios. El sistema de llamadas no está definido así que puede haber más ficheros de inicialización dependiendo de la distribución concreta.

- `/etc/profile`, `~/.bash_profile`, `~/.bash_login`, `~/.profile`: estos ficheros son de configuración general. Se ejecutan por este orden cuando se invoca la shell como login. (La diferencia entre una "login shell" y una "no login" es la forma de invocarlas). Actualmente se tiende a limar esta diferencia entre login y no login, haciendo los ficheros de inicialización equivalentes o incluso haciendo que uno llame al otro.

- `~/.bash_logout`: fichero que se ejecuta cuando acaba una login shell.

- `~/.bash_history`: fichero de inicialización de la historia (ultimas líneas tecleadas) Sirve para poder utilizar la flecha arriba para evitarse teclear y otro atajos (podéis consultarlo con el comando `history`).

Pregunta 2.8: *¿Cómo podéis conseguir que cada vez que abráis una shell desde el entorno de ventanas os aparezca una línea que diga "Passsa tronco."?*

Pregunta 2.9: *¿Que ficheros de la shell* `bash` *de los mencionados anteriormente tenéis en vuestro sistema?*

2.3. El funcionamiento básico de la shell

La shell (cualquier shell) tenéis que verla como un programa que lee la entrada, le aplica una serie de modificaciones según sus propios comandos y a continuación se la pasa al núcleo. Así, por ejemplo, si tecleamos `ls`, la shell busca dentro de todos los subdirectorios listados en la variable `PATH` un ejecutable llamado `ls`, lo encuentra en `/bin` y le dice al núcleo que ejecute el comando `/bin/ls`.

Fijaros que con esta forma de trabajar que es común a todos los *nix, `ls` no siempre tiene que ser un programa que liste entradas en un subdirectorio (aunque normalmente lo será para no hacer las cosas aun más difíciles). Uno de los comandos útiles por si hay problemas con este tema es `which comando` que lo que hace es devolver el ejecutable que realmente se ejecutará cuando tecleemos `comando` en la entrada del `bash`. Por ejemplo, si tecleamos `which ls` la salida será:

```
/bin/ls
```

donde la línea nos dice que efectivamente se ejecuta el programa `/bin/ls`.

¿Es esta la única posibilidad de que un programa modifique su funcionamiento, cambiando el ejecutable? La respuesta es no. Probad a teclear `alias ls`. ¿Que significa la línea de resultado? Esta línea puede variar dependiendo del entorno de inicialización en el que trabajemos (incluso en muchas ocasiones no saldrá). Nos dice que cuando tecleemos `ls` en realidad primero la shell cambia el comando por `ls --color=auto` (este detalle puede variar según la configuración de vuestro ordenador) y a continuación buscará el comando `ls` hasta encontrarlo en `/bin/ls` y a continuación le dirá al kernel que ejecute el programa `/bin/ls` con la opción `--color=auto`. Este sistema de substituciones es recursivo y en realidad en lugar de ser un lio (que también) es uno de los secretos de la potencia de las shells.

El comando `alias` es muy potente a pesar de su sencillez, ya que nos permite personalizar los comandos simplemente añadiéndoles un alias mas cortito. Si tecleamos simplemente `alias` obtendremos una lista de los alias definidos en ese momento.

Pregunta 2.10: *¿Cuántos alias tenéis en vuestro sistema?*

Si tecleamos:

```
alias la='ls -a'
```

tendremos añadido un nuevo alias. Con `alias la` nos imprimirá una sola línea (el `alias` de `la`). Si queremos quitar un alias se usará el comando `unalias`, por ejemplo, `unalias la` o `unalias -a` para quitarlos todos. Habitualmente los alias se definen en los ficheros de inicialización de forma que siempre los tendremos disponibles.

Pregunta 2.11: *¿Cómo haríais para que siempre que hacéis un remove (`rm`) os pida confirmación por defecto antes de borrar nada? ¿Y para que no lo haga?*

Aparte de los alias hay otro sistema que define el comportamiento general de la shell. Este sistema son las variables de entorno (environment), que se suelen definir en los ficheros de inicialización. Para ver las variables de entorno basta con ejecutar el comando `env` que nos da una lista de todas las variables de entorno.

Pregunta 2.12: *¿Cuántas variables de entorno tenéis en vuestro sistema?*

Las variables de entorno se suelen definir en los ficheros de inicialización. Hay muchas y todas se explican en la ayuda de la shell (`man bash`), pero las más usadas son:

- `HOME`: El directorio al que se va a parar cuando se hace `cd` o el directorio por el que se substituye la `~`. Inicialmente coincide con la penúltima entrada de la línea del usuario de `/etc/passwd`.
- `PATH`: Es una lista de directorios separados por ":" donde la shell buscará los ejecutables. Se ejecutará el comando que se encuentre primero si está en varios (por eso puede ser útil `which`).
- `LANG` o `LANGUAGE`: Idioma local a usar si es posible. Es común que empiece por `xx_XX` (las dos primeras letras significan el idioma, las segundas el país).
- `LC_nombre`: Ajustes de localización del parámetro `nombre`.

Para saber el valor de una variable basta con imprimirla `echo $variable`, por ejemplo, `echo $SHELL` imprime la shell que estamos usando. `echo`, como ya hemos dicho, es un comando que imprime por pantalla lo que ponemo a continuación. La shell se limita a substituir `$variable` por el contenido de `variable` antes de decirle a `echo` que imprima. También podemos modificar la variable para que tome otro valor (ojo no todas las variables se pueden modificar): `variable=valor`. Por ejemplo, podemos querer modificar el idioma por defecto: `LANGUAGE=es_ES`. De todas formas no funcionará si los nuevos lenguajes que no están

instalados (como es el caso en los ordenadores en los que estáis trabajando), así que este tipo de cambios es más fácil hacerlo desde el entorno gráfico. Además es importante darse cuenta que una shell es independiente de otra, así que un cambio del valor de una variable, solo vale para esa shell concreta. Si queremos que una variable esté en todas las shells, lo mejor es modificarla en los ficheros de inicialización. Por otro lado, una forma de que las variables de un script sirvan para las shell hijas (cuando se ejecuta un script se crea una nueva shell "hija"que lo ejecuta) es usando el comando `export` o el comando `source`.

Pregunta 2.13: *¿Qué variable de entorno contiene el nombre de usuario? ¿Y el directorio actual?*

Probad a declarar una varible nueva y a imprimirla. A continuación cread un script que imprima dicha variable y cree otra nueva distinta. Ejecutadlo. Imprimid la variable creada en el script desde la línea de comandos.

Pregunta 2.14: *¿Imprime el script la variable que habéis creado en línea de comandos? ¿Y la línea de comandos la variable creada en el script? Esto es porque el script se ejecuta en una* `shell` *independiente ¿Cómo varía el resultado si creais las variables con* `export`*? ¿Y si ejecutáis el script con el comando* `source` *(hacedlo paso a paso y por orden para ver el resultado)*

Pregunta 2.15: *¿Qué diferencia hay entre ejecutar un script con el comando* `source` *o sin él?*

El conjunto de variables de una shell es mayor que sus variables de entorno. Si queremos ver todas las variables podemos teclear `set`. También es bueno saber borrar variables: `unset variable`. Fijaros que no es lo mismo borrar una variable que asignarle un valor vacío. También podemos asignar a unas variables el valor de otras, útil, por ejemplo, para aumentar los directorios del `PATH`:

```
PATH=$PATH:./
```

Pregunta 2.16: *¿Para que es útil la línea anterior?*

Aparte de trabajar con las variables de entorno, en un script podéis aumentar el número de variables simplemente declarando una nueva (es decir, le dais un valor y ya tenéis nueva variable, como en el que hicimos para crear usuarios). Además, en un script siempre tenéis unas variables llamadas `$0`, `$1`, etc que contienen por orden los parámetros del script, y `$#` que contiene el número de parámetros con el que lo habéis ejecutado. Probad el siguiente script (recordad usar la comilla invertida ` que se introduce con la tecla del acento abierto):

```
for i in `seq 0 $#`; do
 echo $i;
done;
```

Pregunta 2.17: *¿Qué hace el script? (Consejo, escribidlo y ejecutadlo con varios parámetros)*

Pregunta 2.18: *¿Y si cambiáis el* `echo $i` *por* `echo $1; shift;`*?*

Pregunta 2.19: *¿Para que sirve la orden* `shift`*? ¿Qué hace el comando* `seq`*?*

Finalmente probad la siguiente combinación:

```
for i in $@; do echo $i; done;
```

Pregunta 2.20: *¿Qué hace este último script? (insisto probadlo con varios parámetros para averiguarlo)*

Fijaros en que vosotros estáis trabajando con la shell `bash` que es la que tiene por defecto Ubuntu. Sin embargo otras distribuciones pueden usar otras shells por defecto (de hecho es una opción del usuario, recordad que era el último parámetro del fichero de usuarios). Los bucles, los condicionales y las variables son característicos de cada shell, así que cuando hacéis un script conviene asegurarse que lo va a ejecutar la shell que toca aunque el script se invoque desde otra. Para ello la primera línea debe contener exactamente:

```
#!/bin/sh
```

Es decir, un sostenido (`#`), una admiración (`!`) y el camino al fichero ejecutable de la shell que queréis que interprete el script. Esto también sirve para scripts en "perl" o "python" (que no son shells sino lenguajes de script). En el ejemplo anterior sería la primera línea de un script que se debe interpretar con la shell Bourne original. En un sistema ya instalado se pueden añadir shells exactamente igual que cualquier otro programa.

Pregunta 2.21: *¿Funcionan los scripts que habéis escrito antes desde las otras shells que tenéis instaladas en el sistema? Para contestar probad a cambiar de shell simplemente ejecutando su nombre y a continuación probad a ejecutar los scripts.*

2.4. Entradas y salidas.

En los entornos Linux todos los comandos que se pueden ejecutar tienen 3 ficheros por defecto con los que interaccionan. El fichero de entrada, el fichero de salida y el fichero de error. Normalmente estos ficheros coinciden con el teclado y la pantalla (recordad que para los sistemas *nix los dispositivos son ficheros). Sin embargo podemos querer modificarlos:

- `>`: El operador > cambia la salida estándar. Así, si hacemos `ls > fichero`, la salida de `ls` se almacenara en el fichero "fichero".
- `2>`: cambia la salida de error estándar. Si hacemos `ls fichero_que_no_existe > fichero` el mensaje de error aparecerá en la pantalla y "fichero" quedara vacío. En cambio si hacemos `2>` el mensaje de error se almacenara en "fichero".
- `&>`: redirige al mismo fichero la salida estándar y la de errores.
- `>>`: añade la salida al fichero en vez de sobrescribirlo.
- `<`: redirige la entrada estándar. Por ejemplo, podemos crearnos un fichero de "yes" con `yes > fileyes` y `^c`. A continuación hacer algo que necesite confirmación: `mkdir prueba, cp * prueba/`. Si volvemos a ejecutar el `cp`, como existe un alias a `cp -i` (pedir confirmación) nos preguntará si ha de sobrescribir todo (si en vuestro ordenador no existe el alias creadlo para probar). Podemos evitar tener que introducir la confirmación haciendo `cp * prueba/ < fileyes` (claro que también, más fácilmente quitando la opción `-i` del `alias` o haciendo `cp -f * prueba/` ;-)

Ojo, si hacéis la prueba con el entorno en castellano deberéis crear un fichero con "sies" en vez de con "yes"... basta con usar el comando `yes s`.

Pregunta 2.22: *¿Para que sirve el comando* `yes`*?*

Un dispositivo interesante a la hora de redireccionar entradas y salidas en *nix es `/dev/null`. Este dispositivo es un agujero negro que simplemente elimina datos (chiste malo de linuxero: `quejas &> /dev/null` :-D). ¿Para que sirve este dispositivo? Fijaros en el siguiente ejemplo:

```
if grep "Error [0-9]*: " $log_file >/dev/null; then
 echo "Ha habido un error"
fi
```

Vamos a intentar ver que hace este ejemplo: el comando `grep` busca en el fichero `$log_file` alguna línea que contenga "Error" y algo más. Si lo encuentra desvía el resultado hacia `/dev/null` (no nos interesa el resultado de `grep`, solo si encuentra algo). Si `grep` ha encontrado algo se cumple el `if` y se ejecuta el `then`. En caso contrario podríamos tener un `else` o lo que sea.

2.5. Las tuberías.

Las tuberías ("pipes" en ingles) son otro de los componentes típicos de los programas de shell de *nix. Las pipes consisten en pasarle la salida de un programa a la entrada de otro, sin crear fichero intermedio. Es decir, son equivalentes a enviar la salida de un proceso a un fichero y leer ese fichero como entrada de otro.

En el caso del ejemplo anterior del sistema de copia, podéis probar con:

```
yes | cp -i * prueba/
```

hará lo mismo que en los casos anteriores pero en este caso sin necesidad de crear un fichero intermedio. Por cierto, también puede interesarnos decir no a todo: `yes n | cp -i * prueba/`)

Probad estos ejemplos para ver mejor como funcionan las pipes (crearos un fichero con texto tonto o usad uno que tengáis por ahí):

```
tac fichero | cat -n
cat -n fichero | tac
cat /etc/group | grep alumne | cat -n
```

Pregunta 2.23: *¿Qué hace cada una de las líneas anteriores? ¿Para que sirve el comando* `tac`*?*

Además de encadenar procesos uniendo sus entradas y salidas, hay otros sistemas de encadenamiento:

- `;`: Ejecuta instrucciones una detrás de otra (lo normal, vamos).
- `&&`: Ejecuta una instrucción y si ha ido bien (devuelve 0) ejecuta la siguiente.
- `||`: Ejecuta una instrucción y si va mal (devuelve un valor distinto de 0) ejecuta la siguiente.

Por ejemplo:

```
cdparanoia -sB "1-" && poweroff
```

(cdparanoia es un programa que sirve para hacer copias de seguridad de vuestros Cds de música en el disco duro). O bien:

```
test -e ficheronuevo || halt
```

Pregunta 2.24: *¿Que hacen las líneas anteriores?*

Las tuberías también son muy útiles para hacer cálculos simples a partir de ficheros que contienen números mediante el comando `bc` que es una calculadora de precisión aleatoria. Empezad probando la siguiente línea:

```
echo "1+2" | bc
```

Pregunta 2.25: *¿Qué hace la línea anterior?*

Ahora probad a crear dos ficheros con números (pueden ser decimales) llamados "bor1" y "bor2" (podéis cambiarles el nombre) y vamos a ver por pasos que hace la siguiente línea:

```
paste -d + bor1 bor2 | bc
```

Pregunta 2.26: *¿Qué hace el comando* `paste`*? (Probad a ejecutar* `paste bor1 bor2` *primero y* `paste -d + bor1 bor2` *a continuación).*

Pregunta 2.27: *¿Cómo haríais para que el sistema multiplicara los números de los ficheros?*

Probad ahora a ejecutar:

```
paste -s -d + bor1 bor2 | bc
```

Pregunta 2.28: *¿Que hace la línea anterior? (Para verlo claro ejecutadla sin el* `| bc`*). ¿Qué instrucción sumaría todos los valores de un fichero?*

Finalmente vamos a ver un ejemplo algo más complejo pero útil (bueno en según que entornos):

```
echo "scale=3;`paste -s -d + bor1 | bc`/`cat bor1 | wc -l`" | bc
```

Pregunta 2.29: *¿Qué calcula la instrucción anterior? (tendréis que averiguar primero que hace* `cat bor1 | wc -l` *y después mirar en el* `man` *de* `bc` *que significa* `scale = 3`*)*

Fijaros, como última observación, que el comando `bc` admite un lenguaje de programación propio y tiene funciones matemáticas predefinidas (como seno o logaritmo) así que permite con mucha facilidad procesar ficheros de resultados.