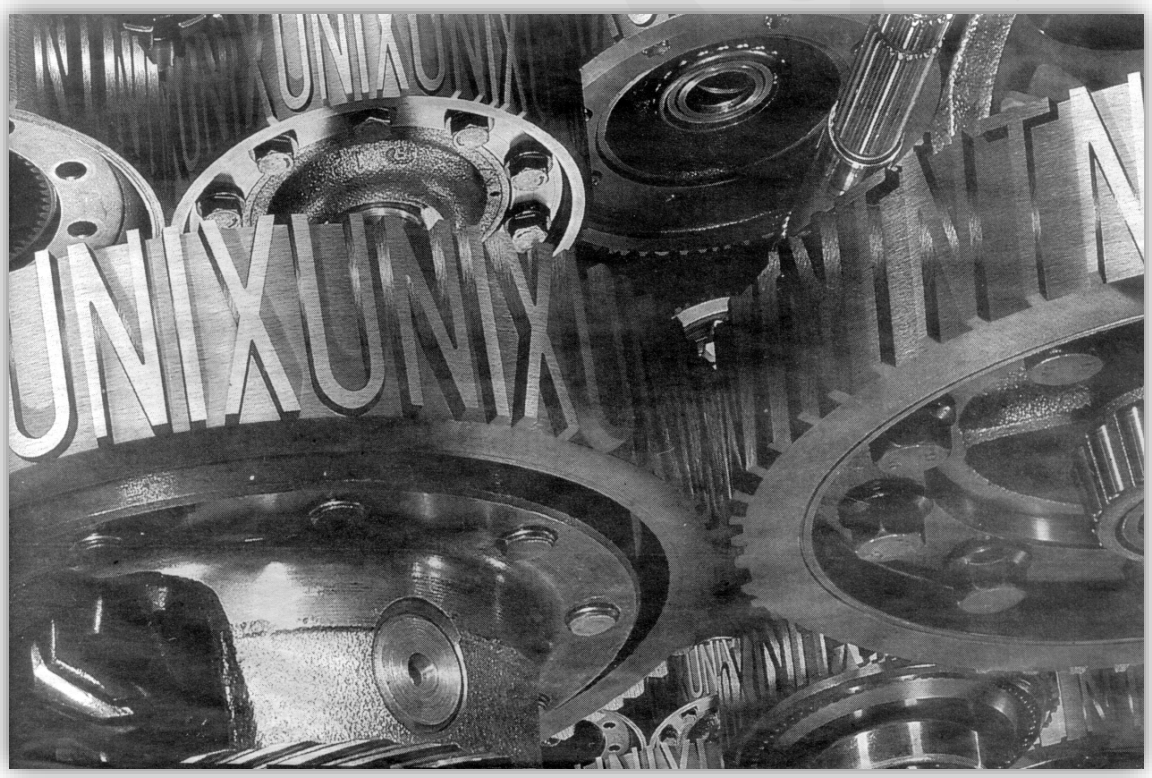
Universidad de Cantabria
Grado de Ingeniería Informática

Sistemas Operativos

Práctica 1 - Partes 1 y 2



Rafael Menéndez de Llano Rozas

Guion

1

Uso de la Shell y del entorno de desarrollo del sistema

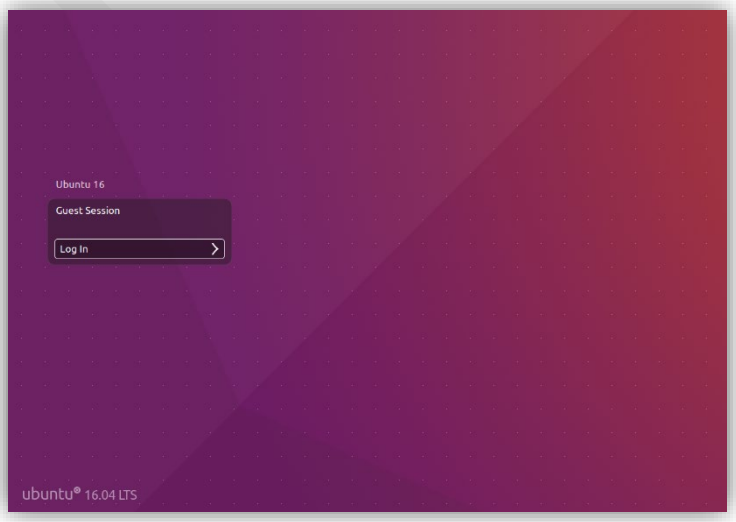
Visión del sistema operativo como usuario: Entrarás al sistema con tu usuario / password y utilizarás los comandos más comunes, así como sus distintas modalidades de uso como comodines, concatenación o desvío. Además, manejarás distintas características de la Shell como la posibilidad de cambiar sus variables o la de realizar macro comandos. Las preguntas que se te hacen en este formato sombreado las deberás responder y anotar para realizar la memoria final junto con pantallazos o código realizado.

Entrada y comandos básicos

Una vez arrancado el computador tendrás que elegir el tipo de sistema operativo con el que trabajar que en nuestro caso será Linux.

A continuación, se te pedirá el usuario/password. Si no te acuerdas de cuál era pregunta al profesor. En esta fase puedes elegir el idioma y el tipo de entorno gráfico que prefieres. Por defecto es Gnome.

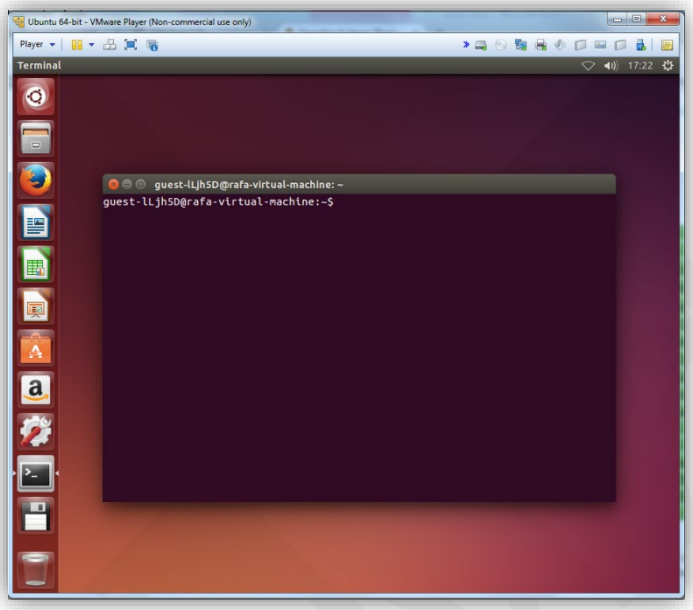
Si estás usando Linux, tienes que tener en cuenta que dispones de varios terminales en uno. La mayoría son de texto y el último es el gráfico (arranque por defecto). Para acceder a ellos tienes que pulsar la combinación Ctrl-Alt-F1 a Ctrl-Alt-F7 (si fueran siete terminales). En el caso de máquinas virtuales la combinación de teclas es distinta (home+F1).



Cuando abandones la sesión (con el comando `exit` o control-d), debes salir de todos los terminales abiertos.

1. Entra al sistema y comprueba si te funcionan los distintos tipos de terminales. Deberás identificarte al menos en uno de ellos. Indica cuál es el gráfico.

Vete al terminal gráfico y explora el entorno, en cada versión puede variar. Por ejemplo, también tienes varios escritorios en uno solo (pruébalo). Para acceder al uso de la Shell debes abrir un terminal que no es más que una aplicación del sistema sin ningún privilegio especial equivalente a los terminales de texto anteriores. Debes saber que Windows también dispone de su propia Shell para administración (Power Shell) y que recientemente se puede instalar una Shell bash de Ubuntu en el propio Windows (probado en 8,1 y 10).



Hay muchos tipos de shell: sh, csh, ksh, tcsh, bash, … Nosotros vamos a emplear la bash.

Todas las shell, una vez están en funcionamiento, esperan la introducción de comandos con un símbolo que se llama inductor (*prompt*), que es configurable como veremos posteriormente. El diálogo con la shell se basa en el uso de comandos que trabajan sobre el sistema de ficheros. Este tiene una serie de propiedades que son las habituales actualmente:

- Es jerárquico, de tal manera que los usuarios pueden agrupar la información relacionada en una unidad y manejarla eficientemente (directorios).
- Aumento dinámico del tamaño del fichero para que contenga el tamaño necesario para almacenar su información, sin necesidad de que intervenga el usuario.
- Ficheros no estructurados, el Linux/UNIX no impone una estructura interna al fichero, por lo que el usuario es libre de interpretar el contenido de los mismos.
- Los ficheros pueden ser protegidos de accesos no autorizados.
- Existe un tratamiento idéntico de ficheros y dispositivos de entrada/salida. Así, los mismos programas pueden utilizar indistintamente tanto ficheros como dispositivos.

La estructura jerárquica es en forma de árbol. Partiendo de un directorio (carpeta en el ambiente gráfico) raíz conocido por "/" del que cuelgan subdirectorios. La separación entre directorios se hace con "/" (a diferencia de Windows que es con "\").

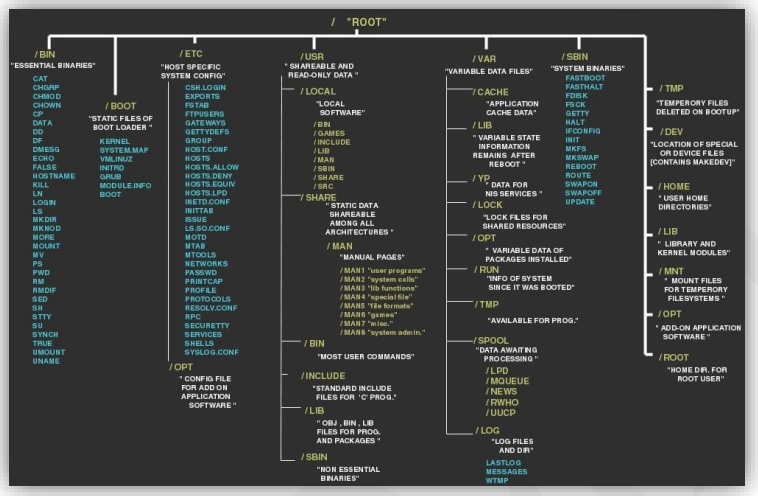
El acceso a esos ficheros se realiza de dos formas:

- Absoluta, partiendo del raíz y dando todo el camino: `/home/pepe/directorio/hola.c`
- Relativa. De forma automática, en cada directorio que se crea, aparecen dos subdirectorios virtuales que son el "." y el "..". Con ellos podemos ir al directorio actual o a su directorio padre.

A diferencia de Windows, en UNIX/Linux no existen unidades nombradas con una letra como A:, B:, C: ... solo existe una unidad principal que mantiene el sistema de ficheros central donde está el directorio raíz y a partir de ese sistema de ficheros, en un proceso de montaje, se solapan los

directorios raíces de las unidades secundarias en directorios (vacíos -si no lo están su contenido se oculta-) de la principal. De esta manera solo existe un único sistema de ficheros, aunque pertenezca a varias unidades.

Además, UNIX/Linux trabaja de la misma forma con ficheros y directorios que con dispositivos. De esta manera, donde se puede poner el nombre de un fichero, también se puede usar un dispositivo.
El árbol del sistema de ficheros puede variar ligeramente de uno a otro sistema, pero existen colgados del directorio raíz una serie de directorios típicos:
El que más nos interesa es el `/home` donde están los directorios de usuarios (cada usuario tiene el



suyo propio con su nombre y está protegido de otros usuarios). Recuerda que en el laboratorio tu directorio estará en `/remotehome` al estar contra un servidor.

Conociendo ya la estructura del sistema de ficheros, podemos trabajar con ella a través de los comandos más habituales (los comandos pueden ser externos –se puede ver su tipo con `file`– o internos –se puede ver con `type`–). Todos los comandos tienen una estructura típica: nombre (en negro), opciones (en verde) y argumentos (en azul), resultado después. En el caso de directorios son:

```
pwd nada nada pinta directorio de trabajo
$ pwd
/usr/ramon/trabajos
```

```
cd nada nada|directorio cambia de directorio (sin arg va al directorio home)
$ cd ../textos
$ cd -                  cambia al directorio de donde veníamos
```

También se puede usar "~" para acceder a home. En Windows se puede obtener con la combinación Alt+126, o mediante la combinación AltGr+4 (el 4 de encima de las letras no el del teclado numérico); en Linux con la tecla AvPág o mediante la combinación AltGr+ñ; en Mac OS X con Alt+ñ.

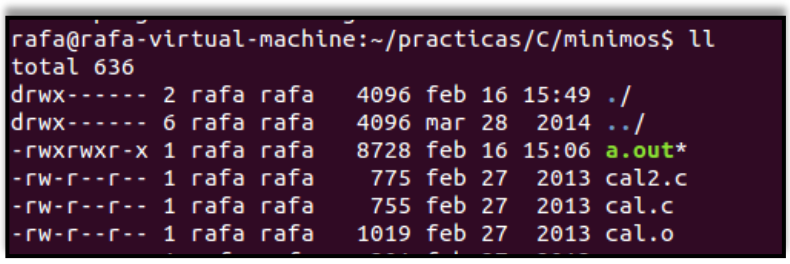
```
mkdir nada directorio hace un nuevo directorio
$ mkdir /usr/ramon/graficos
```

```
rmdir nada directorio remueve (borra) un directorio vacío
$ rmdir graficos
```

```
ls -a,c,l,r,s,... nada|directorio lista directorios
$ ls -alrt          muestra todos los archivos (incluidos los ocultos) de forma
extendida y ordenados por fecha de última modificación en orden creciente
```

En este comando el nombre del directorio es opcional y las opciones más usadas son: `a` para dar todos los ficheros, `t` para ordenar por fecha, `s` para mostrar tamaño, `r` para invertir el orden de salida y `l` para dar formato largo a la salida del directorio, esta opción se usa tanto que hay una orden que sustituye a `ls -l` y es `ll` (si no existe se puede crear como veremos).

En Linux incluso se pueden presentar los distintos tipos de ficheros por colores (verde para ejecutables, azul oscuro para directorios blanco para ficheros regulares, hay más),



donde en total nos dice el número de bloques de espacio listados (636), después en la siguiente línea y en el primer carácter nos dice el tipo de fichero que es (por ejemplo d para directorios o – para ficheros regulares), y a continuación los permisos del mismo (lectura, escritura o ejecución), el número de enlaces, el propietario del fichero, el grupo al que pertenece, el tamaño en bytes, la fecha del último cambio y por último el nombre del fichero.

Y para ficheros:

```
cp varias f1 f2 fn directorio copia ficheros origen destino
$ cp origen destino
$ cp f1 f2 f3 directorio

cat sin ficheros (concatenar) muestra un fichero
$ cat pepe.c
main ()
{
...

more varias ficheros muestra por páginas un fichero
$ more pepe.c
```

tiene su propio lenguaje de comandos, principalmente se puede dar un número positivo (+) o negativo (-) de pantallas para ir hacia adelante o hacia atrás, si se da el número sin signo nos posicionamos en una determinada pantalla, si pulsamos `[enter]` avanzamos una pantalla adelante, con b iremos para atrás o con espacio una línea para adelante y si pulsamos q nos saldremos del programa, (en algunos sistemas UNIX se utiliza el comando `pg` de similares características).

Existe la versión `less` más avanzada que no necesita leer el fichero completo para presentarlo.

```
less varias ficheros muestra por páginas un fichero
$ less pepe.c
```

también tiene su propio lenguaje de comandos (además del `more`): ctrl+u página arriba, ctrl-d página abajo, ctrl-p línea arriba, ctrl-n línea abajo, g al principio, G al final, y / para buscar.

```
head varias fichero muestra las primeras líneas de un fichero
$ head pepe.c
main ()
...

tail varias fichero muestra las últimas líneas de un fichero
$ tail pepe.c
...
}

mv varias origen destino mueve o renombra ficheros a un directorio
$ mv registro almacen
$ mv reg1 reg2 reg3 muestras

ln varias 4 formas enlaza (link) un fichero a otro o a un directorio
$ ln /usr/ramon/ventas /usr/ramon/graficos  (3ª forma)
```

Primera forma: dos nombres de ficheros (crea un enlace con otro nombre en el mismo directorio), segunda forma: un único nombre (crea un enlace en el directorio actual de un fichero en otro directorio), tercera forma: fichero y directorio (crea un enlace en el directorio indicado), cuarta forma (*target*) con –t

directorio fichero (equivalente a la 3ª solo que se pone antes el directorio). En este comando la forma de enlazado se puede hacer de dos maneras: *duro (*enlace *hardware*, duro o físico) es el por defecto, crea una "puerta" a un fichero existente, pero solo existe un fichero y las dos puertas tienen los mismos privilegios; *lógico (*enlace *software*, blando o simbólico), se crea un fichero especial (acceso directo) que simplemente guarda información sobre cómo llegar al original (se usa –s y se puede leer con `readlink`). Los directorios solo se pueden enlazar de esta manera. El número de enlaces aparece al hacer un `ls -l`.

```
rm -i fichero elimina un fichero
rm -r directorio elimina un directorio y sus ficheros
$ rm fich
$ rm -r /usr/ramon/ventas     Borra recursivamente sin preguntar
$...
```

Hay que tener mucho cuidado con este comando, al igual que con `rmdir`, ya que no existe la vuelta atrás y si se borra no se podrá recuperar lo eliminado (no hay papelera).

Siempre que tengamos alguna duda sobre el uso de un comando se puede usar este otro comando:

```
man [seccion] comando
```

Nos presentará la información que hay disponible sobre un determinado comando o aplicación. La información del manual está dividida en secciones y deberíamos saber a qué sección pertenece lo que intentamos buscar (si es un comando normalmente la 1, que es tomada por defecto), pero también podríamos buscar una llamada al sistema (la 2) u otra entidad del sistema. El formato de salida será el del comando `less`. Entre las muchas opciones que tiene, la -k es posiblemente la más interesante y en Linux es equivalente al comando `apropos` que busca el *string* indicado en la base de datos de búsqueda del sistema. Hay que destacar que es diferente a -K que busca el *string* en todo el manual y puede ser muy lento. Un comando para buscar de forma sumarizada en que sección está algo es `whatis`.

```
man -k comando ⇔ apropos comando   /     whatis comando
```

Las secciones del comando suelen residir en el directorio `/usr/man` y son: 1. Comandos de usuario. 2. Llamadas al sistema. 3. Funciones y rutinas de librería. 4. Configuración y formato de ficheros. 5. Miscelánea. 6. Ficheros especiales y hardware. 7. Ficheros especiales y hardware. 8. Comandos de mantenimiento. 9. Drivers.

Ahora que ya conocemos el comando `man` (suele ser interesante colocar otro terminal para el `man`), podemos describir la arquitectura de los comandos y como vienen descritos en él:

- **[ ]** indica que hay argumentos opcionales.
- **…** indica que se pueden expresar múltiples valores del argumento.
- **|** indica que los argumentos a la izquierda o derecha son excluyentes.
- **{ }** indica argumentos mutuamente exclusivos pero uno obligatorio.

Debes saber que hay cientos de comandos, la gran mayoría comunes a todas las Shell. Una referencia para la `bash` la puedes encontrar por ejemplo en *http://es.wikipedia.org/wiki/Comandos_Bash*.

2. Vete a tu directorio de usuario, compruébalo con `pwd`. Crea un directorio a partir de tu directorio llamado temporal, entra en él, y compruébalo de nuevo con `pwd`. Intenta hacer lo mismo desde el directorio `/remotehome` ¿Puedes hacerlo? Di el porqué. Averigua si `cd` es interno o externo. ¿Y `pwd`?

3. Ve a ~/temporal y con el comando `touch` (si no sabes que es prueba el `man`) crea estos ficheros: "pepe1", "pepe2", "pepe3", "pepe12" y "pipe3". Copia, mueve y enlaza de dos formas (hardware y software) algún fichero de `temporal` en su directorio padre. Si haces el enlace blando de forma relativa ¿qué ocurre? ¿Cómo lo solucionarías? Explora las opciones de los comandos. Localiza en qué sección están del manual.

4. Haz un `ls` con diversas opciones (a, l, s, r, y R) y describe las diferencias. Cuando termines borra el directorio `temporal` con `rmdir`. Indica si puedes hacerlo o no. Si es que no, ¿Cómo lo harías (pero no lo hagas)?

Los comandos que hemos introducido en una sesión de trabajo siempre se pueden recuperar (se guardan en el fichero `.bash_history`), en lo que se conoce como historial (lo que simplifica el trabajo diario) con el comando `history`:

```
$ history
```

Las flechas arriba y abajo nos permitirán recorrer el historial. Otra forma de aprovecharlo es usando el comodín "`!`" seguido de alguna de las letras por la que empezaba el comando almacenado, recuperarlo sin necesidad de escribirlo de nuevo. La búsqueda se hace en sentido inverso, es decir, se busca primero el más reciente. También se puede usar `control+r` (`ctrl+r` presionar la tecla control y r) seguido de los caracteres de algún comando anterior para que la *Shell* lo localice.

También se pueden autocompletar las líneas de comando usando el tabulador. Si existen ambigüedades nos lo dirá y deberemos continuar escribiendo. Sabed que `ctrl+a` nos mandará al comienzo de una línea y `control+e` al final, `alt+f` y `alt+b` mueven adelante y atrás una palabra, `ctrl+k` y `ctrl+u` borran desde el cursor hasta el final o hasta el principio, y `alt+d` y `ctrl+w` borran la palabra de adelante o de atrás. El uso de estas facilidades hace que se queden en tu memoria "muscular". Se pueden cambiar con `set –o vi` / `set –o emacs` si estás acostumbrado a otros editores.

Por último y para acabar con este apartado, podemos encadenar varios comandos en una sola línea de comandos simplemente separándoles con un ";". También se pueden encerrar varios comandos entre paréntesis para ejecutarlos dentro de una *subshell*:

```
(cd; pwd) ; pwd      pintará el directorio home, y después el actual, dejándonos en él.
cd ; pwd ; pwd       pintará dos veces el directorio home y se quedará en él.
```

Existen otras variantes de cadenas de ejecución, pero en estos casos condicionales. Cada vez que se ejecuta un proceso, devuelve a la *Shell* un valor lógico (*exit status*) que la indica si este proceso se ha ejecutado bien o no. El estatus valdrá cero (al contrario que en lenguaje C) cuando el comando se haya ejecutado con éxito. Se puede utilizar este valor para encadenar la ejecución de varios comandos (date cuenta que esto no es canalización, se verá después) de forma condicional, de tal manera que, si se ha ejecutado el primero bien, se ejecute el segundo o no. Para ello existen dos operadores principales: el AND, &&, y el OR, ||:

```
comando1 && comando2      se ejecutará el 2 si el 1 es correcto (status 0)
comando1 || comando2      se ejecutará el 2 si el 1 es no correcto (status <> 0)
```

5. Usa el comando `history` y el comodín "!". Prueba el `ctrl+r` (búsqueda reversa) y el resto de combinaciones de teclas para editar líneas. Comprueba como cuando se hace uso de los paréntesis, se ejecuta lo encerrado en una *subshell*. Prueba el concatenado condicional para copiar un fichero que exista o no, si es así, que se edite. Haz lo mismo compilando un programa con y sin errores y ejecutándolo.

Comodines

Una característica que está en otros sistemas operativos es la posibilidad de utilizar caracteres especiales que sirven de comodines (*Pattern Matching*) para referenciar un grupo de ficheros.

En la *Shell* existen dos caracteres especiales el "*" que sirve para sustituir a una secuencia de 0 o más caracteres y el "?" que solo sustituye uno. A diferencia de otros sistemas operativos el punto en un nombre de un fichero no actúa como un carácter especial separando nombre y extensión, sino que forma parte del nombre completo, por lo cual el "*" también lo sustituye.

Algunos ejemplos de utilización son:

```
rm * borrará el directorio actual.
rm pep* borrará todos los ficheros que sean o empiecen por pep
rm reg? borrará los ficheros reg1, reg2 y reg3, pero no reg12
```

Hay una modificación del "?" y es cuando se especifican entre corchetes los caracteres que se van a sustituir, así:

```
rm reg[12] solo borrará los ficheros reg1 y reg2
```

También se pueden utilizar otros caracteres especiales: { } para replicar (expansión) un grupo de nombres encerrados entre las llaves y separados por comas:

```
a{b,c,d}e se referirá a abe, ace y ade
```

Además, en la `tcsh` existe el carácter '^' (94) para negar cualquier selección hecha con los comodines. Por ejemplo, si tengo los ficheros `bang, crash, cruch, y out`, `ls ^cr*` dará como resultado `bang out`.

Por último, existe un carácter especial más que es el '\', que antepuesto, sirve para interpretar literalmente (*quoting*) algún carácter especial como puede ser un comodín o un espacio. Esto es útil cuando por error (por estupidez o capricho) hemos introducido uno de estos caracteres en el nombre de un archivo e intentamos acceder a él, piensa, por ejemplo, como podrías borrar un fichero que tiene en medio del nombre el carácter espacio. También se utiliza para escribir una orden en varias líneas, acabando cada una de estas con un '\' para que la *shell* sepa que no termina.

6.	Vete a temporal y prueba con los ficheros creados los caracteres comodines ("*", "?", "[ ]"). Utiliza también el comodín de negación (es necesario ejecutar la `tcsh`). Crea un fichero con un espacio en su nombre y bórralo. Vete a tu directorio con "~". Usa el comando `echo` con un argumento que no quepa en una línea (uso de \).

Redirección y Canalización

Otra característica importante de las *Shell* es la redirección de los canales de entrada/salida estándar. Cuando se ejecuta un programa, normalmente la *Shell* le adjudica al proceso tres canales o ficheros estándar, por un lado, un canal de entrada (`stdin`), de donde el programa toma los datos; por otro lado, un canal de salida (`stdout`), donde se dejan los resultados; y por último un canal de errores (`stderr`) que recibirá los mensajes de error que se produzcan durante la ejecución del proceso (como curiosidad, en la *Shell* hay hasta 9 canales). Los dos primeros canales están asignados normalmente al terminal donde se trabaja (teclado y pantalla) y el tercero a la pantalla, pero es distinto de forma lógica al canal de salida. Estos tres canales tienen 3 descriptores de fichero asignados: 0 para entrada, 1 para salida y 2 para errores.

Desde la línea de comandos se puede cambiar la asignación a estos canales a través de un proceso que se llama redirección y que utiliza dos caracteres especiales el ">" y el "<", el primero para la salida y el segundo para la entrada, estos símbolos pueden ser precedidos por el descriptor de fichero, aunque si son los normales (0 y 1) son omitidos, no así el de error.

Si tenemos un programa que nos pide datos desde la línea de comandos y produce un resultado en la pantalla, podemos hacer que esos datos los tome desde un fichero en vez de escribirlos uno a uno con el teclado, para ello utilizaremos la redirección de entrada:

```
$ programa < datos
```

Donde `datos` es un fichero de texto con la información necesaria para que el programa funcione. De igual manera podemos hacer que la salida de ese programa no se pierda en la pantalla, sino que se escriba en un fichero, entonces tendremos que utilizar la redirección de salida:

```
$ programa > salida
```

e incluso podemos hacer que tome los datos de un fichero y los saque en otro:

```
$ programa < datos > salida
```

Cada vez que usamos la redirección de salida, si el fichero existe se sobrescribe (podemos cambiar este comportamiento con una opción de la *Shell*: `set -o noclobber`). Si queremos añadir los nuevos datos usaremos ">>":

```
$ programa >> salida
```

Y como hemos dicho, la equivalencia entre un fichero y un dispositivo en UNIX/Linux es total, por lo que podemos redirigir la entrada o la salida hacia un dispositivo (en el directorio `/dev` están representados todos los dispositivos del sistema) como la impresora o nulo:

```
$ programa > /dev/lp     $progr > /dev/null
```

Incluso si no queremos que aparezca nada se puede eliminar la salida con el dispositivo especial `/dev/null`, que hace que todo lo que le llegue desaparezca.

También se puede redirigir la salida de error con "2>":

```
$ cat pepe 2> juan       $cat pepe > aaa 2> bbb
```

de tal manera que si no existe el archivo `pepe` se escribirá el error en el archivo `juan`.

También se puede hacer que la salida de errores y la normal (o viceversa) vayan hacia el mismo canal. Para ello existe la combinación &> o >&, aunque se recomienda la primera forma. En el primer caso (salida normal desviada a errores) deberíamos usar 1&>2, en el segundo caso 2&>1. En las *shell* modernas como bash solo se usa &>:

```
$ programa &> fichero
```

Es conveniente saber que las redirecciones que hemos utilizado aquí son para las *shell* derivadas de `sh` directamente: `ksh` y `bash`. Para la `tcsh` también sirven con dos salvedades: que no se usan los descriptores de ficheros, lo que implica que la salida de errores se redirecciona con ">&·"; y que la salida normal y la de errores no se pueden usar simultáneamente, con lo cual tenemos que utilizar la ejecución en una sub-shell para corregirlo:

```
$ (programa > salida ) >& error
```

7. Usando la redirección de salida crea un fichero `calendario` con el comando `cal`, primero de 2017 y después añadiendo el 2016. Mira el contenido del fichero con `cat`, con more y con `less` usando sus subcomandos (retorno, espacio, b, q, etc.).

8. Usa el comando `ls -l` con el directorio `/root` (o algún otro que no tenga permisos) y desvía la salida hacia un fichero ¿sale algo por pantalla? ¿A qué es debido? ¿Cómo se podría evitar que salga? Desvía de forma unificada tanto la salida normal como la de errores en un único fichero.

La canalización está relacionada con la comunicación entre procesos y aunque se utilice también en la línea de comandos y junto con la redirección, no tiene nada que ver con esta. La canalización consiste en transmitir datos directamente de un proceso (comando o programa) a

otro (la salida del primero sirve de entrada al segundo) para lo que se utiliza el símbolo "|" (*pipe*, *pipeline* o tubería) que los comunica. Por lo tanto, mientras la redirección comunica procesos con ficheros/dispositivos, la canalización comunica programas. No hay límite entre el número de procesos que se pueden encadenar:

```
$ programa1 < entrada | programa2 | programa3 > salida
```

Hay dos comandos especiales relacionados con la canalización. Uno es `xargs` que coge la salida estándar y la entrega a otro comando que se pone como argumento. El otro es `tee` que hace que lo que pasa entre dos programas se guarde en un fichero:

```
$ echo 'pera limón manzana' | xargs mkdir      (creo tres directorios)
$ programa1 | tee fichero | programa2          (en fichero lo que va de 1 a 2)
```

La canalización se puede combinar con la concatenación de comandos condicional como se ha hecho en los dos ejemplos siguientes:

```
echo hola|write rafa||mail rafa        Manda hola a un usuario o bien por pantalla
                                       si está en el sistema o bien por mail.
cat fich|write usuario&&echo si        Manda un fichero por pantalla a un usuario y
                                       pone si solo si realmente funcionó.
```

También se puede unificar la salida de dos programas con llaves (se debe acabar cada orden con "; " y empezar con " " ojo con los espacios que son ␣).

```
{␣echo comienzo;␣cat fichero;␣} | wc -l     unifica la salida de los dos primeros
```

Hay comandos que no leen nada de la entrada estándar (`ls`) y otros de tratamiento de texto que usan mucho con la canalización: `head` para mostrar las primeras líneas de un fichero, `tail` las últimas, `nl` para poner números de líneas, `wc` cuenta caracteres, palabras y líneas, `uniq` para eliminar líneas repetidas, `sort` para ordenar líneas (se puede usar –n y –r), `cut` para trocear líneas, `fmt` formatea texto a un número de caracteres por línea, etc. Existen muchos más.

9. Realizar un pipeline (tubería) con el comando `echo`, `tee` y `wc`. Haz una concatenación de comandos con llaves del comando `echo` y `more` de un fichero y entúbalo al comando `nl`, date cuenta de la diferencia de usar y no usar llaves.

Expresiones regulares y búsquedas

Algunos comandos del sistema (`awk`, `ed`, `find`, `grep`, `sed`, `vi`…) funcionan como motores de búsqueda y usan lo que se conoce como expresiones regulares (pueden ser básicas BRE o extendidas ERE) para indicar lo que estamos buscando (no confundir con comodines –metacaracteres de shell– ya vistos). Una **expresión regular** es un conjunto de reglas que se emplean para especificar uno o más elementos dentro de una cadena de caracteres. Veamos algunos de los más usados (hay bastantes posibilidades, ver http://www.gnu.org/software/grep/manual/grep.html#Regular-Expressions):

- Cualquier carácter **individual** con "**.**". Ejemplo l..a puede ser lapa, lana, laca, …
- **Conjuntos** de caracteres con "**[ ]**". Ejemplo expresi[oó]n puede ser expresion o expresión. Se puede usar el carácter "-" para expresar rangos de caracteres. También hay algunos predefinidos: [[:alnum:]], [[:alpha:]], [[:cntrl:]], [[:digit:]], [[lower:]], [[:space:]], [[:upper:]], etc. Por ejemplo [[:alnum:]] es cualquier alfanumérico.
- **Anclajes**: Comienzo con "**^**". Ejemplo ^x todo lo que empieza por x. Fin con "**$**". Ejemplo n$ todo lo que acabe en n.
- **Repetición**. Lo anterior a "*****" estará repetido 0 ó más veces. Ejemplo xy* puede ser x, xy, xyy, xyyy, … Lo anterior a "**?**" es opcional y estará repetido una vez. Ejemplo, xy? Puede ser x o xy. Lo anterior a "**+**" estará repetido 1 ó más veces. Ejemplo xy+ puede ser xy, xyy, xyyy, … Con las llaves "**{n}**" lo precedente n veces (número exacto). Con "**{n,}**" n o más veces, con "**{,m}**" al menos m veces, con "**{n,m}**" al menos n veces pero no más de m.

- **Grupos**. Con "**( )**". Ejemplo (xy)+ sería xy, xyxy, xyxyxy, …
- Las expresiones pueden ser alternativas con "|". Ejemplo x|y|z es x o y o z.
- Secuencia de **escape** con "\". Sirve para dar un significado especial al carácter que viene detrás. Por ejemplo: \. \$ * \s (será el espacio) \S ((cualquier cosa menos espacio).

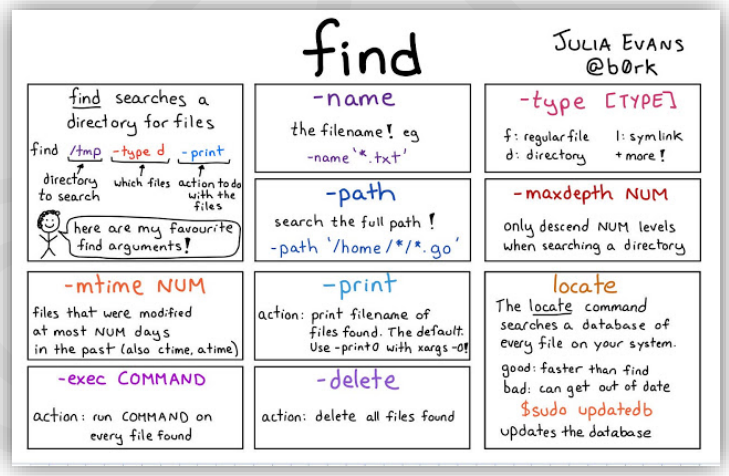
Si las usamos con `grep` para construir patrones de búsqueda, estos suelen estar encerrados entre apóstrofes (o comillas). `grep` puede tomar opcionalmente un argumento que es un fichero donde buscar, o sin él, hacer la búsqueda en la entrada estándar. Con la opción `-s` no ponemos mensajes de error, con `-m n` paramos después de n ocurrencias, con `-r` buscamos recursivamente… Si queremos usar ERE debemos usar la versión extendida –`E` o `egrep`). Hay una tercera versión para los familiarizados con **perl** `-P`. Ejemplos:

```
ls | grep 'pepe[123]'                          Busca las ocurrencias de pepe1 2 y 3
ls | grep 'pepe[[:digit:]]'                    Lo mismo pero todos los números
egrep '(192.168.0.30|192.168.0.40)' test       Busca las dos cadenas en test
egrep '192.168.0.[30,40] ' test                Busca las que tengan 30 o 40
egrep '192.168.0.[0-2]0' test                  Busca entre 0 y 2
egrep '^hostname' test                         Las que empiecen por hostname
egrep 'node$' test                             Las que terminen en node
```

10. Crea un fichero con el comando `cal`. Busca dentro del fichero creado con el comando `grep` las ocurrencias de la palabra "Febrero".

11. Realizar un pipeline (tubería) con el comando `ls` (en forma recursiva del directorio /) y el comando `grep` para buscar los ficheros que tengan "pepe" en su nombre y acaben en un dígito.

Hay una forma de búsqueda en el sistema de ficheros y ejecución combinada de lo buscado, se realiza con el comando `find`, muy útil para usuarios y sobre todo para labores administrativas.



La búsqueda siempre se realiza de forma recursiva salvo que se indiquen directorios donde buscar como primer argumento opcional. Toma, como resto de argumentos, expresiones que pueden ser de tres tipos: opciones, test y acciones. Dada la complejidad del comando es mejor ver algunos ejemplos y revisar el `man` para otros usos:

1. Buscar un fichero por su nombre:

```
a. find  -name abc.c
b. find  -name abc*              // con comodines
c. find  -iname aBc*             // sin tener en cuenta mayusculas
d. find  -not -name *.php        // invirtiendo criterio
e. find  -type f -name abc*      // con tipo fichero
f. find  ./d1 ./d2 -name abc*    // en dos directorios solo
```

2. Buscar por alguna cualidad (los permisos se verán en otro apartado):

```
a. find  -perm 0664              // con ese permiso
b. find  ! -perm 0777            // sin ese permiso
c. find  -perm /u=r              // permiso de lectura
d. find  -user luis              // archivos de luis
e. find  -mtime 50               // de los últimos 50 dias
f. find  -atime 7                // accedidos los últimos 7 dias
g. find -mtime +30 -mtime -60      // entre 30 y 60 dias
h. find -cmin -60             // cambiados –estatus- hace 60 minutos
i. find -mmin -60               // modificados hace 60 minutos
j. find -amin -60               // accedidos hace 60 minutos
k. find . –size +40M            // ficheros de más de 40 MB.
l. find –user pepito            // buscar ficheros de pepito
```

3. Buscar y realizar acciones (`exec`) con lo encontrado, también se puede usar `xargs`:

```
a. find -type f -name "*.txt" -exec rm -f {} \;    // los borra
b. find  -perm 0777 -print -exec chmod 644 {} \;   // cambia permiso
c. find /tmp -type f –mtime +7 | xargs –0 rm
d. find ~/ -name '*.mp3' | xargs -0 mv /mp3
```

Advierte como si usas la tercera versión con `exec` debes respetar el curioso formato del comando `{}␣\;` (ojo al espacio ␣) que indica que se ejecute lo obtenido exclusivamente en lo encontrado por `find`. La canalización se usa de forma alternativa con el comando `xargs`.

> 12. Averigua los cinco ficheros más grandes de todo tu directorio de dos maneras diferentes: usando `find` y usando directamente `ls`. Pista: deberás usar los comandos `sort` y `head` canalizados.

Hay otros dos comandos que se usan para localizar ficheros, el primero es `locate`, usa una base de datos donde busca el fichero y por tanto es más rápido pero esa base de datos debe estar actualizada, para lo cual se usa `updatedb` como superusuario y el segundo `whereis` que busca comandos y nos dice la ubicación, los ficheros fuentes y su manual (`-bsm`).

```
$ locate pepe1
$ whereis ls
```

Ejecución en segundo plano (background)

Hay dos formas de ejecutar un programa: una es la usual en primer plano "*foreground*" y la otra es en segundo "*background*". En la primera, el programa una vez ejecutado toma el control de la entrada/salida, en la segunda el control vuelve a la Shell desde donde lo hemos ejecutado (hay que tener en cuenta que las salidas del programa se podrán mezclar en el terminal con el *prompt* y lo que escribamos a la Shell). Para realizar la ejecución de esta manera basta con poner un "&" detrás del comando:

```
$ programa > salida &
```

Cuando se ejecutan varios procesos en *background* se puede ver si realmente se están ejecutando con el comando `ps` (estatus de proceso) que tiene varias opciones que nos dicen que procesos representar (todos los del sistema, los de un usuario, etc.) y su formato. La forma más sencilla es:

```
$ ps
PID TT TIME COMMAND
3243 30 0:11 bash
3254 30 0:01 ps
```

la información que aparece se refiere a la identificación del proceso (PID), al número de terminal, al tiempo empleado en la ejecución y al nombre del proceso.

El directorio `/proc` (pseudo sistema de ficheros de solo lectura) hace de interfase con el núcleo del sistema (más información con `man proc`). En él aparecen una serie de "directorios" con nombre numérico (PID) que corresponden a los procesos activos en ese momento en el sistema. En estos directorios podemos encontrar toda la información de un proceso, como por ejemplo los ficheros que tiene abiertos (`fd`), el mapa de memoria (`maps`), o el estado del mismo (`status`).

Otra forma de ver la jerarquía de los procesos es el comando `pstree` que nos da una imagen en forma de árbol de los procesos que se están ejecutando actualmente.

Conociendo el PID de un proceso siempre lo podemos abortar con el comando `kill`, al cual daremos como argumento ese PID. En Linux muchas veces resulta más cómodo dar el nombre del proceso (o procesos si tienen el mismo nombre como `a.out`) que queremos eliminar, esto se puede hacer con el comando `killall`. En ambos casos los `kill` pueden tomar una opción numérica que es la señal (evento del sistema) que queremos enviar, en el caso de no poner nada, la señal enviada es la número 15 (se pueden ver todas con `kill -l`). Los procesos pueden estar protegidos contra la recepción de señales, en ese caso el comando `kill` no les afectaría. Por seguridad hay una señal que nunca puede ser evitada que es la número 9 (`SIGKILL`). Por eso en muchos casos se usa:

```
$ kill -9 12038      o      $ killall -9 proceso
```

Los procesos son generales al sistema, pero la *shell* permite tratar a los procesos creados en una sesión de una forma más sencilla, son los *jobs*. No hay que pensar que los *jobs* son diferentes a los procesos, simplemente es otra forma de utilizarlos y no tener en cuenta al resto de procesos creados en el sistema. De esta manera, nosotros podemos suspender un proceso que se esté ejecutando con `ctrl-z` (no confundir con ejecución en *background*, que si se esta ejecutando en otro plano) y al tener el control de la *shell* de nuevo, mandar un nuevo proceso a ejecutar en *foreground* o *background*. Para ver la lista de procesos de la sesión, *jobs*, tenemos el comando del mismo nombre `jobs`, que nos dará una lista de los mismos entre corchetes:

```
$ jobs
[1]   Detenido                 vi
[2]-  Detenido                 vi
[3]+  Detenido                 vi
[6]   Hecho                    ls --color=auto -R &> /dev/null
```

Para ejecutarlos solo tenemos que utilizar los comandos *`fg`* y *`bg`*, que ponen respectivamente al *job* en ejecución *foreground* o *background*. El `+` indica el que se toma por defecto, el `-` el siguiente.

```
$ bg 3
$ fg 1
```

13.	Indica qué dos procesos están siempre cuando se ejecuta ps. Averigua las opciones del comando `ps`. Ejecuta en *background* el editor de textos `vi`. Mira si el proceso se ha creado realmente. Si es así mátalo (`kill`). Repítelo con ejecución en *foreground* y para al proceso (debes saber cómo hacerlo). Después, pásalo de nuevo a *foreground*, páralo de nuevo, pásalo a *background* (indica la diferencia entre este estado y el anterior) y finalmente mátalo con `killall`. Piensa en una forma de averiguar la shell que estás ejecutando.

Fin de la primera sesión

Ambiente multiusuario

Como se ha dicho, el sistema UNIX es multiusuario y multiproceso, esto significa que varios usuarios pueden estar ejecutando varios procesos a la vez. Para poder hacer esto, el sistema tiene que tener unas determinadas características:

- Sistema de seguridad de acceso. Se debe garantizar que un usuario tiene a salvo su trabajo libre de accesos no permitidos. En el caso del sistema Linux/UNIX a través de un password.
- Seguridad de ficheros. Se debe garantizar que los usuarios no accedan y por tanto cambien o destruyan ficheros del sistema o de otros usuarios. En Linux/UNIX se utilizan tres tipos de permisos y tres modos de acceso que se verán posteriormente.
- Compartición de recursos. Los recursos comunes deben ser repartidos equitativamente entre todos los usuarios (ver capítulo de gestión de recursos). En el Linux/UNIX se utilizan sistemas de *spooling.*
- Control de recursos. También se debe llevar la cuenta de los gastos de los distintos usuarios (uso de CPU, papel, etc.). Se llama *accounting*.
- Administración de sistemas. El sistema debe garantizar mecanismos para hacer *backups* de los trabajos de los usuarios y la posible recuperación de errores.

Como hemos dicho, la seguridad del sistema se lleva a cabo en dos niveles, por un lado pidiendo el *password* al usuario (cuando *root* ha creado un usuario con el comando `adduser` o `useradd`) le pone una palabra de paso temporal en `/etc/passwd` —actualmente se usa /etc/shadow por seguridad— que debería ser cambiada[1]), y por otro lado y una vez dentro del sistema, no permitiendo el acceso a ficheros no autorizados. El *password* es una palabra secreta que solo conoce el propietario de la misma y que debe introducir en el sistema cuando entra al mismo. Para proteger el secreto cuando se introduce esta palabra, el terminal se pone en blanco de tal manera que los caracteres introducidos no se vean reflejados en la pantalla. El usuario siempre puede cambiar su *password* a través de un comando del sistema operativo. En nuestro caso no conviene hacerlo ya que son usuarios por red.

El segundo nivel de seguridad es el acceso a los ficheros. Cada fichero del sistema, incluyendo los directorios, tienen un usuario propietario y por tanto un grupo (cada usuario pertenece a un grupo). De esta manera, el acceso al mismo dependerá de si se es el propietario, de si se es del mismo grupo de usuarios, o de si se es de otro grupo cualquiera. Y es el propietario del fichero el que puede cambiar estos tres permisos de entrada. Los grupos se pueden crear por razones de utilidad (proyectos, intereses comunes, mismo nivel de jerarquía, …) o por razones de acceso hardware (quien puede acceder al sonido, al CD, al *bluetooth*, …).

A la vez existen tres tipos de acceso al fichero: para leerlo (r), para cambiarlo (w), o para ejecutarlo (x); (en el caso de un directorio hacer un `ls`, incluir un fichero dentro o hacer un `cd` a él) pudiéndose cambiar estos accesos a cada tipo de usuario. Teniendo en cuenta esto, existen en total 9 permisos que el propietario del fichero puede cambiar y que se pueden ver al hacer un listado de un directorio en modo largo, es decir, con la opción -l (recuerda que puede tener un alias ll):

```
-rw-rw-rw- 2 grupo 7 alumnos 3654 Nov 24 12:17 fichero
 uuugggooo
```

El primer trío de la izquierda rwx corresponde a los permisos del usuario (uuu), cuando aparezca una letra significará que se tiene ese permiso y cuando aparezca un "-" que no, los otros dos tríos son los del grupo (ggg) y los de los otros usuarios (ooo).

La primera letra que falta indicará, como ya hemos visto, el tipo de fichero, siendo el "-" el correspondiente a un fichero regular y "d" a un directorio (existen más tipos).

[1] En la página os he dejado un manual básico de administración para que lo tengáis de referencia.

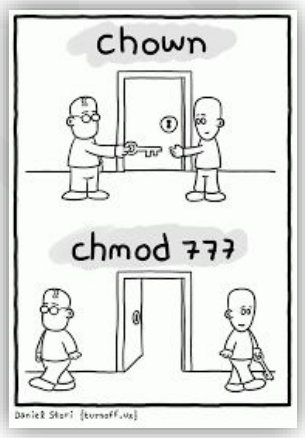
Los permisos de un determinado fichero se pueden cambiar a través de la orden `chmod` (cambio de modo), que es un poco especial ya que no tiene opciones, pero si los permisos que se quieren cambiar, y toma como argumentos los ficheros o directorios afectados. La forma de poner los permisos se compone de tres partes, por un lado, a qué tipo de usuarios afecta el cambio, simbolizados por tres letras: la u para el usuario, la g para el grupo y la o para los otros usuarios, por otro lado, si se da o se quita el permiso, simbolizado por + ó -, y por último el tipo de permiso de lectura (r), escritura (w) y ejecución (x). Así, para quitar los permisos de lectura y escritura a todos menos el propietario del fichero *nominas* habría que hacer:

```
chmod go-rw nominas
```

Otra forma de utilizar este comando es dando como permiso un número octal de tres cifras, donde cada cifra es un tipo de usuario y donde cada dígito binario del octal es un tipo de permiso, por ejemplo 777 (sería 111 111 111) daría todos los permisos a todo el mundo. Si quisiéramos hacer ejecutable una macro y poder ejecutarla sin `sh`, podríamos hacerlo de dos formas:

```
chmod +x macro
chmod 755 macro
```

Hay que tener en cuenta que el *inodo* del fichero contiene 12 bits de control de acceso de los cuales 9 corresponden a estos permisos. Los otros 3 bits son conocidos como *setuid*, *setgid* y *sticky bit*. El primero de ellos servirá para que un ejecutable tome como usuario el propietario del ejecutable y no al usuario que lo ha ejecutado. Por ejemplo el fichero `/etc/passwd` solo puede ser editado por el *root*, sin embargo cuando ejecutamos `passwd` lo podemos cambiar, esto es debido a que `passwd` tiene activado el *setuid*. De forma análoga para el grupo. El tercer bit (el bit pegajoso) se usa también en ejecutables cuando queremos que permanezcan permanentemente en memoria secundaria. Actualmente en Linux solo se utiliza en directorios cuando queremos que solo el propietario pueda renombrar/borrar los ficheros. Hay que tener en cuenta que la manipulación de estos bits es peligrosa ya que ciertos agujeros de seguridad se basan en su manipulación.



14.	Elige un fichero (o créalo) y cámbiale los permisos de lectura y escritura para que nadie pueda acceder a él. Intenta mostrar su contenido por pantalla, ¿qué ocurre?

Por defecto, todos los accesos están permitidos, a no ser que esté definido algún otro acceso por el comando **`umask`**. Este comando toma como argumento una máscara (código de inhibición) que es restado al 666 (acceso total sin ejecución ya que los ejecutables lo tienen activado por defecto) cada vez que se crea un fichero (no aplicable a directorios). Normalmente se ejecuta cuando arranca una shell `umask 022` lo que equivale a unos permisos 644 (`rw-r--r--`).

Como hemos dicho todos los ejecutables tienen activados los bits de ejecución, por lo que no son necesarias las extensiones tipo .exe o .com. Date cuenta que también se pueden poner permisos de ejecución a las macros por lo que ya no sería necesario anteponer el "`sh`".

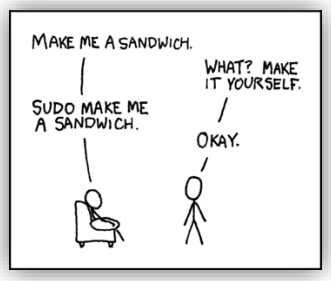
Está claro el significado de los permisos en los ficheros regulares, pero ¿qué significa en los directorios?:

- Ejecución: permite acceder al directorio (cd).
- Lectura: permite listar el contenido del mismo.
- Escritura: permite la creación de nuevos ficheros en su interior.

Para poder aplicar los permisos, en la meta información del fichero (se ve con `ls` en formato largo) debe aparece el propietario y el grupo al que pertenece el fichero. Tanto propietario (usuario) como grupo de usuarios, están codificados por un entero que se llama UID y GID.

Estos se pueden cambiar con los correspondientes comandos:

```
chown [-R] nuevo_usuario fichero (Cambia el UID).
chgrp [-R] nuevo_grupo fichero (Cambia GID).
```



En la distribución de Ubuntu en vez de crear dos usuarios (uno el *root* y otro el personal de usuario) como es recomendable, han optado por solo crear un usuario por defecto. Cada vez que ese usuario tiene que hacer alguna labor de administración, tiene que anteponer el comando `sudo` a la instrucción (pide el *pasword*) para que funcione. En otras distribuciones se usa el comando `su` para cambiar a administrador temporalmente. Otros comandos para manejarse en el sistema multiusuario son:

```
who  y  w            para decir que usuarios están en el sistema.
su usuario           para cambiar temporalmente de usuario.
login usuario        para cambiar de usuario sin retorno.
passwd [usuario]     para cambiar el password de un usuario.
```

Si queremos comunicar usuarios que estén en ese momento en el sistema, usaremos el comando `write usuario` (este comando se puede confundir con la llamada al sistema `write`, de ahí la necesidad de `whatis` y las secciones del manual ya vistas). Con `write` conseguiremos que le aparezca en la pantalla a otro usuario el mensaje que le hemos escrito (terminará en control-d). Si no queremos que nadie nos escriba (puede ser molesto), usaremos `mesg n`. Esto se hace modificando el acceso (`rw`) del dispositivo que estamos usando de terminal.

```
write usuario [terminal]    para mandar un mensaje a otro usuario
mesg y, n                   para permitir o no que nos escriban
```

Hay una forma más sofisticada de comunicarse que es el comando `talk usuario` que divide la pantalla en dos partes una para el emisor y otra para el receptor (no instalada en el sistema). La tercera forma de comunicación off-line sería el conocido `mail`.

Si la comunicación es entre máquinas distintas se usa el protocolo `rsh`, actualmente `ssh` para arrancar una Shell remota en otro computador. Hace falta que en el destino exista el servicio (demonio) de `rsh` - `ssh`.

```
rsh [-l usuario] host.maquina.com "mkdir testdir"
```

15. Crea un terminal sobre ventana y uno de texto. Comprueba que estás en el sistema con `who` (`who am i`). Utiliza el comando `write` contigo mismo entre los dos terminales. Prueba `mesg`. Con el comando `ps` mira el dispositivo que estás usando en la shell. Observa el contenido del directorio `/dev` hasta encontrarlo. Usa la redirección para mostrar con `cat` el contenido de un fichero en el dispositivo que usas (puede ser `tty` o `pts/x`) de forma equivalente a como lo harías con un fichero. Hazlo también para que el contenido salga en el otro terminal.

Hay una excepción en la utilización de los ficheros en los sistemas UNIX, ya que existe un encargado del mismo que tiene privilegios sobre el resto de los usuarios, es el llamado superusuario. Por convención, el nombre del superusuario es *root* y como este nombre indica su

directorio de trabajo es el directorio `/root` y tendrá los suficientes privilegios como para acceder a cualquier fichero del sistema, lo único que no puede conocer es nuestro *password*. Cuando el *root* quiere comunicarse con los usuarios utiliza el comando `wall`.

Por último, para compartir los recursos comunes del sistema entre varios usuarios, en concreto la impresora, lo que se utiliza es el mecanismo de *spooling*, de tal manera que cuando varios usuarios intentan acceder a la impresora a la vez, este mecanismo almacena los requerimientos e impone un orden de envío a la misma (se forma una cola de impresión).

Para utilizar el spooling existen varios comandos en UNIX, los más usados son:

```
lp ficheros      para enviar a la impresora un fichero(s) (lpr en LINUX).
lpstat           para ver la cola de impresión (lpq en LINUX).
cancel trabajo   para cancelar un envío anterior (lprm en LINUX).
```

Las variables de la Shell

Otra característica potente de las *shell* es la definición de variables que servirá para guardar datos del usuario y también para modificar el comportamiento de la propia *shell*. Esto se hace o bien a través de la creación de variables de usuario, o bien modificando el valor de las definidas por la propia *shell*.

La definición de las variables depende del tipo de Shell. En el caso de tener la *Shell* `bash` se utiliza su nombre seguido de "=" y el nuevo valor. Hay que recordar que en Linux/UNIX es distinto una letra mayúscula que una minúscula y que las variables se suelen poner en mayúsculas. Si el nombre tienes espacios se debe encerrar entre apóstrofes. Una vez definida la variable, se podrá utilizar en cualquier comando del sistema simplemente anteponiendo a su nombre el carácter "$" (expansión de variable). También se puede poner su nombre entre llaves y si hay espacios con comillas simples o dobles.

Para ver variables en la *Shell* se utiliza el comando interno `echo` (también existe `printf`). Por ejemplo (ten en cuenta que en los ejemplos que te pongo el *prompt* está simbolizado con "$", no confundirlo ahora con el "$" de la expansión de variables) podemos definir y ver variables:

```
$ DIRECTORIO=/home/pepe/proyecto/sesion/trabajo/grupo
$ DIRS='. ~'
$ echo $DIRECTORIO $DIRS
$ echo mis directorios ${DIRS} definidos
```

En cuanto a las variables definidas por la *shell*, las más usadas son (hacer `man bash` para ver todas también con el comando `set` se pueden ver incluso las definidas por nosotros):

```
PATH     Donde se guardan los directorios donde se pueden encontrar ficheros
ejecutables.
         PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
HOME     El directorio de trabajo (home) del usuario.        HOME=/home/rafa
PS1      El string que se utiliza como inductor prompt.       PS1='\w \$'
USER     El usuario.                                          USER=rafa
PWD      El directorio donde estamos.                         PWD=/home/rafa
TERM     Tipo de terminal desde el que estamos conectados al sistema. TERM=xterm
SHELL    Tipo de shell que ejecutamos.                        SHELL=/bin/bash
```

En el primer caso la definición de la variable será el conjunto de directorios donde la *shell* va a buscar ejecutables (en otros directorios no se podrá ejecutar nada) separados por ":". Antiguamente se ponía también el directorio "." para poder ejecutar en cualquier directorio, pero era un riesgo de seguridad y se quitó, por eso estamos obligados a poner "./" antes del ejecutable. La tercera, PS1, es el *prompt* o inductor, alguno de los modificadores que existen (hacer `man bash` para ver todos) para variar su valor son:

```
\d          La fecha.
\H          El nombre de máquina.
\j          El número de jobs.
\s          El nombre de la shell.
\t          La hora.
\u          El usuario.
\w          El directorio actual.
\!          El número del comando dentro de la historia de comandos.
\#          El número de comando.
\$          Será $ en caso de usuario normalo # en caso de root.
```

un caso habitual puede ser: `[\u@\h \W]\$`, que pondría como *prompt* el usuario@ordenador, directorio de trabajo y el $ de siempre.

Debemos tener en cuenta que la definición de una variable solo tiene efecto en esa sesión de *shell*, si queremos que sirvan para *shell* o programas creados a partir de ella, hay que utilizar el comando `export` (realmente es un comando interno [built-in] de la *shell*) para convertir variables de *shell* en variables de entorno o *environment variables*. Esto es importante para realizar macros (*Shell scripts*) ya que las macros se ejecutan en una sub-shell; o programas (desde un programa en C se puede tener acceso a las variables de entorno como tercer argumento de *main*).

Para ver las variables de entorno se utiliza el comando `printenv o env` que nos pondrá en pantalla las que tenemos asignadas (si solo queremos conocer una podremos hacer `echo $variable`). Para borrar una variable se utiliza el comando `unset`.

Si necesitáramos realizar operaciones con las variables, se pueden asignar valores con el comando interno `typeset –i` (i para *integers*):

```
$ typeset –i numero=1
```

En el caso de la `tcsh`, para crear variable se utiliza `set` y para ver las variables de entorno el comando setenv.

16. Si estás ejecutando una `bash`, mira el valor de las variables de shell `PS1` y `PATH` utilizando `echo` y averigua si son de entorno con una canalización y `grep`. Arranca una shell de tipo `tcsh` (si está instalada) y mira el valor esta vez, si están indefinidas piensa el porqué. Sal de la `tcsh`. Mira todas las variables que tienes definidas, averigua la shell que estás usando y el tipo de terminal. Cambia el valor de PS1 para que te ponga un mensaje personal terminado en $.

17. Crea una variable de shell para asignar el nombre de un directorio a esa variable, comprueba su contenido y después úsala con el comando `cd`. Comprueba con grep si existe como variable de entorno, conviértela a entorno y comprueba de nuevo. Por último, elimínala.

El comando `set` también se utiliza para ver y cambiar las opciones definidas en la *Shell,* por ejemplo `set –o` pondrá las opciones actuales y para ponerlas o quitarlas se usa `–o` o `+o` (ojo + para quitar), algunas ya las hemos visto como `set –o emacs` o `vi` para recupera y editar el histórico de órdenes.

Ficheros de configuración

Cuando abrimos un terminal, sobre él se arranca automáticamente una Shell y al empezar se ejecutan una serie de macros que definirán su comportamiento. Estas macros están en el directorio *HOME* de cada usuario y empiezan por punto para estar ocultas.

Normalmente la secuencia de carga es:

- bash:
  `/etc/bashrc` → `/etc/profile` → `$HOME/.bashrc` → `$HOME/.bash_profile`

- tcsh o (csh):
  `/etc/csh.cshrc` → `/etc/csh.login` → `$HOME/.tcshrc(.cshrc)` → `$HOME/.login`

Pero en algunos sistemas esto no es así por tanto siempre conviene revisarlo, por ejemplo pueden no existir los ficheros de usuario, pero haber un `.profile`.

Las variables de Shell solo están definidas en la sesión de trabajo, si queremos crear una variable nueva o modificar las existentes cada vez que arranquemos la shell, debemos definirla o redefinirla en esos ficheros. Otra tarea habitual dentro de estos ficheros es redefinir comandos con `alias`.

Por ejemplo, podríamos hacernos nuestro propio comando `ls` como `dir` o abreviar la escritura de `ls –l` con `ll`.

```
$ alias cd.. = 'cd ..'      para incluir la eliminación del espacio
$ alias dir = 'ls –l'       para tener un dir de MSDOS
$ alias ll = 'ls –l'        para abreviar ls –l
```

Si quisiéramos eliminarlos utilizaríamos el comando interno de la *Shell* `unalias`.

Para que los efectos del cambio actúen de inmediato, sin necesidad de salir y entrar de la Shell, se puede ejecutar el comando `source`.

18. Averigua los ficheros de configuración que tienes ocultos en tu directorio personal. Haz una copia por seguridad y examina el aspecto del que tienes que modificar. Modifícalo (al final) para cambiar la variable adecuada para redefinir tu inductor (*prompt*). Comprueba si hay alias definidos. Si no es así crea un alias que se llame ll, si existe crea uno que te parezca útil.

Creación de macrocomandos (macros): shell scripts

Un aspecto importante de las shell de UNIX, es que se pueden programar y hacer macrocomandos compuestos de la ejecución secuencial de otros comandos más simples. Es lo que se llama en UNIX *shell scripts* (guion) o normalmente macros. Esta característica es muy útil cuando se utiliza a menudo una secuencia de comandos, ya que nos permitirá repetirla fácilmente con un solo nombre. Como se ha visto en teoría esto constituye el actual procesamiento *batch*.

Básicamente, un macrocomando es un fichero de texto que puede tener el atributo de ejecutable (`chmod`) o puede ser ejecutado con `sh macro` y en el que hemos escrito una serie de comandos que queremos ejecutar secuencialmente, puede incluir redirección, canalización y *background*. La estructura de las macros puede llegar a la misma complejidad que la de un programa de alto nivel, de hecho, dentro de una se pueden utilizar las mismas estructuras de control (selección e iteración) que en un programa y también se puede hacer uso de variables, tanto definidas por el usuario como por la propia *shell*.

Una vez escrita la macro con el editor de textos, podremos ejecutarla, para ello tendremos que arrancar una mini-shell `sh` (Bourne) a la que le damos su nombre. Cuando termine de ejecutar

la macro, también terminará la nueva sub-shell. Otra forma de ejecutarla es cambiando el modo del fichero haciéndolo ejecutable, en este caso se utiliza una *Shell* como la invocante. Otra forma es poniendo punto para que no se llame a otra *Shell*.

```
$ sh macro    si es que hemos llamado a la macro macro
$ . .bashrc   es útil para ejecutar los ficheros de configuracion
```

Además, una macro puede recibir datos desde el exterior en la propia línea de comandos, estos datos son conocidos como argumentos posicionales y pueden ser usados dentro de la macro antecediendo su posición con el símbolo $ (ya que realmente son variables), de tal manera que $0 es el nombre de la macro, $1 el primer argumento, etc. No es usual poner más de 10 argumentos, aunque si se diera el caso, se podrían trasladar con el comando interno `shift`.

```
#Si empieza por # es comentario/Esta macro se ha ejecutado asi: sh macro ar1 ar2
ar3
echo "el valor del primer argumentos es $1 "
echo "el valor del segundo argumentos es $2 "
echo "el valor del tercer argumentos es $3 "
shift
echo "el segundo argumento es $1 y el tercero $2  "
```

Las variables más comunes son (antecediéndolas el $ para obtener su valor):

```
$1 a 9   Variables de argumentos posicionales.
$0       El nombre de la propia macro.
$#       El número de argumentos dados.
$?       El código exit del último comando ejecutado, 0 sin errores.
$!       El identificador de PID del último proceso ejecutado.
$$       El identificador de la Shell ejecutante
$*       Un string conteniendo todos los argumentos posicionales.
```

Dentro una macro se puede acceder a las variables de entorno y también definir variables locales y leerlas desde teclado. Para ello se usa el comando interno `read` (si hacéis un `man` de `read` os daréis cuenta de la importancia de poner la sección adecuada en el `man`, ya que existen dos, el perteneciente a la shell y la llamada al sistema `read`, ojo, en sistemas Ubuntu actuales no aparece).

El siguiente ejemplo nos demuestra cómo crear variables y utilizar comandos dentro de una macro (un # indica que la línea es un comentario):

```
# Esta macro usa variables y los comandos echo y read
echo "Por favor entra tu apellido"
echo seguido de tu nombre:
read nombre1 nombre2
echo "Bienvenido a UNICAN $nombre2 $nombre1"
printf "otra forma de pintar \n" ; echo "otra forma de pintar"
```

El comando `echo` (existe también el comando `printf` en bash) sirve para poner un mensaje en pantalla, tanto con comillas como sin ellas, `read` leerá desde teclado los valores de variables, en este caso `nombre1` y `nombre2` y por último se pintarán estos valores usando el $ delante de la variable. Una línea se puede separar en dos usando ";" .

Si nos interesa especificar la *Shell* con la que se va a ejecutar la macro y que no coja una por defecto, como primera línea se puede poner `#!` y el nombre absoluto para llegar:

```
#!/bin/sh     #!/bin/bash    …
```

También se pueden hacer evaluaciones numéricas (recuerda el comando `typeset`, será más rápido) y operar con las variables con el operador doble paréntesis "`(())`". La *Shell* maneja valores enteros por defecto, si queremos trabajar con decimales se usa el comando `bc`. Los operadores numéricos son: " `+ - * / %` ".

Veamos un ejemplo de evaluación de variables (recuerda no usar espacios en asignaciones a variables, es erróneo poner EDAD = 22, debe ser EDAD=22):

```
EDAD=22                    o   typeset -i EDAD=22
MAYOR=$EDAD+25             el valor será literal
echo $MAYOR
MAYOR=$(($EDAD+25))        el valor será numérico
printf "$MAYOR \n"         otra forma de pintar
echo $((1+1))
echo 3/4|bc -l             otra operacion echo "1.2 * 2.4" | bc -l
COTA=$(echo "10*0.9" | bc -l)
```

Sabiendo usar variables, la entrada/salida y la evaluación, pasamos al siguiente aspecto que son las sentencias selectivas e iterativas. Como en C, el primer paso para hacerlo es poder construir expresiones **condicionales** y lógicas, para ello tenemos el comando `test`. Los valores de las expresiones condicionales son contrarios a los del lenguaje C, en la *shell* 0 es TRUE y distinto de 0 es FALSE, esto es debido a que cuando un programa funciona bien devuelve un valor de *exit* de 0 que puede ser obtenido con la variable $?. `test` adopta la forma de cualquier comando: el nombre, las opciones y los argumentos (más información con `man test`), las opciones nos dirán lo que tenemos que comprobar (testear). En lo que se refiere a ficheros como argumentos, las principales opciones son: -e para existencia, -d para directorio y -f para fichero regular (hay muchas más http://wiki.bash-hackers.org/commands/classictest).

```
test -e $fichero    Comprueba si el fichero $fichero (variable) existe o no.
```

En la mayoría de las *Shell,* en una macro en vez de utilizar `test` se puede usar `[ ]` (¡ojo con los espacios!) que es equivalente: `[ -e fichero ]`:

```
[ -e $fichero ]   Comprueba si el fichero $fichero (variable) existe o no.
```

También admite argumentos de tipo cadena y enteros sobre los que puede establecer comparaciones. Para la `bash` (ojo con los espacios):

```
(cadenas)  =  !=
(números) -eq -ge -ne -gt -le -lt
```

Y poder usar expresiones lógicas entre ellos: AND (-a), OR (-o) y NOT (!)

```
[ cadenas  = otra_cadena ] -a ![ tercera != cuarta ]
[ entero -eq otro entero ] -o [ otro -ne otromas ]
```

Además, si usamos la `tcsh` (es la única ventaja de usar esta *Shell*), en las expresiones también se pueden utilizar los mismos operadores lógicos que en el lenguaje C e incluso se pueden utilizar operadores aritméticos. Los más importantes para `tcsh` son:

```
== != <= >= < > || && !        | & ^ ~ >> << + * - / % ( )
```

Las expresiones construidas con `test o [ ]`, o la ejecución de cualquier comando (que como hemos visto también es una expresión lógica) son utilizadas para realizar las sentencias de selección e iteración:

```
if [ expresión ]
then
     código si 'expresión' es verdadera
fi
```

Existe la variante de ponerla sentencia de selección como `if [ expresión ]; then`.

Unos ejemplos de utilización serían:

```
FILE=~/.bashrc        # definicion de variable
if [ -f $FILE ]       # si es fichero
then
   echo el fichero $FILE existe
fi

if test -e $FILE       # otra forma
then
   echo el fichero $FILE existe
fi

echo $#              # pinto los argumentos pasados a la macro
if [ $# -eq 0 ]      # si no hay
then
     printf "no has pasado argumentos \n"
fi

if [ rafa = rafa ]; then  echo si; fi

# ejecutado con dos argumentos: sh macro argu1 argu2
#dentro de la macro argu1 es $1 y argu2 es $2, $0 es el nombre macro
echo $1 y $2
if [ ! $1 = $2 ]          #o if ! [ $1 = $2 ]
then
   echo son distintos
fi
```

En este caso estamos usando el comando `test` combinado con la sentencia `if` (terminado con `fi`). También lo hacemos con `[ ]`. Comprobamos el número de argumentos numéricamente y miramos si dos cadenas son distintas con `not`.

Como ocurre en un lenguaje de alto nivel las sentencias se pueden anidar y puede haber `if` dentro de `if`.

También existe la combinación de la sentencia `if` con la `else`, al igual que estas estructuras se pueden anidar (`else if` o su contracción `elif`) su estructura sería:

```
if condicion
then
     comandos (condición es true)
else
     comandos (condición es false)
fi
```

Se pueden hacer comprobaciones más complejas utilizando los comandos existentes en la *shell*, ya que sabemos que la ejecución de un comando devuelve un código exit que es lógico:

```
#!/bin/sh    # se ejecuta con la sh
#USER=rafa
if who | grep $USER > /dev/null         # no me interesa la salida
then
    echo $?                             #pondrá 0
    echo $USER esta en el sistema       #si funciona el grep
else
    echo $?                             #pondrá 1
    echo no esta                        #si no funciona
fi
```

En este caso utilizamos el comando `who` (del ambiente multiusuario) combinado con el comando de búsqueda `grep` para saber si un usuario está en el sistema, la salida estándar se desvía a `null` para que no aparezca en pantalla.

Existe también la construcción `case`, que compara una palabra dada con los patrones de la sentencia, como siempre termina con la sentencia inversa `esac`:

```
case palabra in
     patron1) comando(s)  ;;
     patron2) comando(s)  ;;
     patronN) comando(s)  ;;
esac
```

Ejemplo:

```
#!/bin/sh
echo $1
case $1 in
       [0-9] ) echo "numero" ;;
       [a-z] ) echo "minuscula" ;;
       "" ) echo "no hay argumento" ;;
       * ) echo "fallo de argumento" ;;
esac
```

También existen sentencias iterativas (bucles), la primera de ellas es la sentencia `for` que utiliza una variable de índice que se puede mover entre los valores de una lista de valores, por ejemplo $* (todos los argumentos posicionales) que será tomado por defecto.

La estructura es (como en el `if`, la primera línea también puede acabar en `; do`):

```
for var in lista de palabras     # ejemplo 1 2 3 4 5 o uno dos tres
do
     comandos $var
done
```

Un ejemplo de utilización del bucle `for` se ve a continuación, donde pasamos a una macro una serie de nombres de usuarios para saber si están en el sistema o no están:

```
for i in $*    #se podría poner for i solamente en el método abreviado
do
     if who | grep $i > /dev/null
     then
          echo $i esta en el sistema
     else
          echo $i no esta
     fi
done
```

La lista de nombres también se puede obtener de la ejecución de un comando con “$( )” o con “``” (apóstrofes inversos o graves) como en estos dos ejemplos:

```
for i in $( cat num ); do  #lo cogera de lo escrito en num
     echo item: $i
done

for i in `ls`              #lo cogera de los ficheros del directorio
do
     echo fichero: $i
done

for i in `seq 0 9`         #commando seq o for i in 0 1 2 3 4 5 6 7 8 9
do
     echo entrada$i
done
```

19. Las macros también funcionan igual en línea de comando, separando cada línea por ";". Crea con una única línea (y un bucle `for`) cinco ficheros vacíos que vayan de vacio1 a vacio5.

Otros tipos de bucle son el `while` y el `until` cuyo significado es claro (uno de permanencia otro de salida) y cuya estructura aparece a continuación:

```
while comando
do
    comandos
done
```

```
until comando
do
     comandos
done
```

En los siguientes ejemplos hacemos un bucle infinito y podemos esperar a que un usuario salga del sistema (`sleep` 60 parará la ejecución durante un minuto):

```
while :          # : es un comando interno que devuelve TRUE (0)
do
     echo siempre
done
```

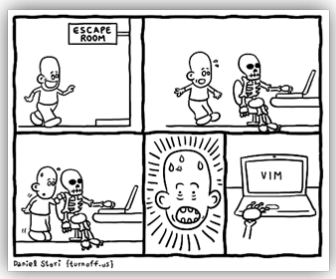
```
while who |grep $1 >/dev/null
do
     sleep 60
done
echo "$1 se ha ido"
```

También se pueden emplear expresiones numéricas (en muchos sistemas no existe en comando interno `let` pero se puede sustituir por:

```
limite=5
i=0
while [ $limite -ge $i ]
do
     echo Acción $i ejecutada
     #let i=$i+1
     i=$(($i+1))    #sustituye al let de arriba
done
```

Para acabar el apartado de las macros, debemos saber que también podemos tener *arrays* de variables y hacer funciones (no vale para sh):

```
#!/bin/bash
#definicion de la función
function funcion ()
{
   echo "argumentos funcionan igual que en las macros: $@"
   echo "Me han pasado esto: $1 $2..."
   echo "Esto es todo en la funcion"
   return 0
}
#definicion de un array de 6 elementos
mi_array=(uno dos tres cuatro cinco seis)
# imprimir el primer elemento, ${ } se llama
expansión y sirve para cualquier variable
echo ${mi_array[0]}
for i in "${mi_array[@]}";
do
    echo "$i"
done
#llamo a una función con dos argumentos
funcion pepe juan
```



20. Edita con `vi` un fichero de texto. `vi` o `vim` es un editor de texto de pantalla básico, tienes un pequeño manual en la página. El uso de `vi` a pesar de ser un editor primario, está justificado por su carácter universal y no depender de las características de un teclado particular. Al menos una vez en la vida es necesario su uso. Incluso, si te acostumbras a él, será más eficiente que cualquier otro editor gráfico).
Ese fichero será una macro que servirá para saber el tamaño en Kb o Mb de una serie de ficheros de un directorio (pista mira el comando `du`). La macro tomará estos ficheros por argumento y pedirá el directorio con lectura por teclado. Solo mostrará el tamaño de los ficheros si existen y si está definida una variable MENSAJE puesta a SI. La variable MENSAJE en un caso estará definida desde fuera de la macro y en el otro caso de forma interna ¿puedes acceder a la variable interna definida en la macro desde la shell?

Certificaciones (solo información)

Debes saber que existe un instituto el LPI (Linux Professional Institute ─ http://www.lpi-spain.es ─) que ofrece certificaciones profesionales que pueden ser de ayuda para el mercado laboral. LPI tiene dos tipos de certificaciones: **Linux Essentials** y **LPIC** (1,2,3):

- **Linux Essentials** está pensado para novatos y aquellos que quieran comenzar su andadura en el mundo del Sofware Libre. Al aprobar el examen se obtiene el "LPI Linux Essentials Professional Development Certificate".
- **LPIC** Estas certificaciones LPI han sido diseñadas para certificar la capacitación de los profesionales de las Tecnologías de la Información usando el Sistema Operativo Linux y herramientas asociadas a este sistema.

Ha sido diseñado para ser independiente de la distribución y siguiendo la Linux Standard Base y otros estándares relacionados. Estas certificaciones LPI están orientadas al puesto de trabajo a desempeñar utilizando para ello procesos de Psicometría para garantizar la relevancia y calidad de la certificación.

Actualmente existen tres niveles de certificación profesional:

- LPIC-1 Linux Server Professional
- LPIC-2 Linux Network Professional
- LPIC-3 Linux Enterprise Professional (especialidades 300, 303 y 304)

Volviendo al Linux Essentials que es el que está relacionado con esta parte, comprende cinco tópicos (en inglés https://www.lpi.org/study-resources/linux-essentials-exam-objectives/):

1. The Linux community and a career in open source
2. Finding your way on a Linux system
3. The power of the command line
4. The Linux operating system
5. Security and file permissions

El primero es literatura que se puede estudiar a solas, el segundo, el tercero y el quinto los has visto en las sesiones precedentes y el cuarto está dedicado al hardware.

Fin de la segunda sesión