4. La Shell

4.1. Shell

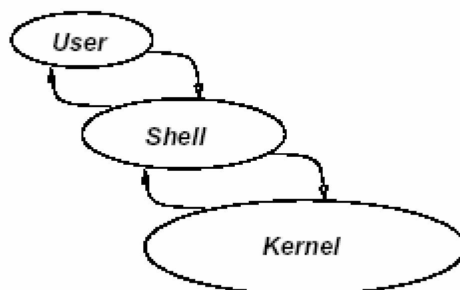
Propiedades

La shell es el programa que más se utiliza en un sistema UNIX, ya que cada vez que se establece una sesión de trabajo, se ejecuta una shell. Este programa permite la comunicación con el sistema, dándole comandos que ejecutar y por último es el que controla el final de la sesión. En el caso de Linux, una vez arrancado el sistema, tendremos varias consolas disponibles (como seis terminales), accesibles a través de la combinación de teclas alt-F1,..., alt-F6 (en algunos sistemas la combinación es de tres teclas: control–alt F1, ... , F6), todas ellas son consolas en modo texto donde se arrancará una shell como las que vamos a ver. Existe una tercera con F7 que es de tipo gráfico.

Las características principales de la shell son las siguientes:

- Permite el dialogo interactivo entre el usuario y el sistema a través de una serie de comandos definidos.
- Permite ejecutar programas en background, es decir, después de empezar la ejecución de un programa, la shell devuelve el control al usuario para que pueda realizar otras tareas. Cuando el primer programa termina la shell avisa al usuario a través de un mensaje que aparece en pantalla.
- Permite la redirección de la entrada y la salida normales por terminal hacia otros dispositivos o ficheros.
- Permite hacer que la salida de programas pueda servir como entrada de otros programas (pipes) y así de una manera sencilla construir programas más complicados.
- Se permite el uso de caracteres comodines para seleccionar uno o más ficheros.
- Se permite programar la shell para hacer comandos como combinación de la ejecución secuencial de otros comandos, es decir, hacer macros, que en el entorno UNIX se llaman Shell Scripts. Las estructuras sintácticas que se pueden usar en una macro dependen de la shell que se utilice, en la C Shell es parecida al lenguaje de programación C.
- Se permite controlar el funcionamiento (características) de la shell, a través de unas variables definidas en la misma. De esta manera podemos cambiar por ejemplo la forma del prompt o el camino (pathname) para encontrar ficheros ejecutables.

La utilización de la shell como se ha visto es bien sencilla, el sistema pone una pantalla de login (de entrada de un usuario) pidiendo que se introduzca el nombre del usuario y su palabra de paso (password) para protegerle de otros usuarios. Una vez que se ha realizado con éxito la entrada, se ejecuta una shell (se puede escoger la shell a ejecutar y además puede haber varias ejecutándose), que pone el prompt en la pantalla y espera que se introduzca una línea de comando, la cual analizará y ejecutará (un programa es código que reside en un fichero y puede ser ejecutado, un proceso es un ejemplar de un programa ejecutándose en memoria) en caso de estar correcta o nos devolverá un mensaje de error en caso contrario. En ejecución normal el control del terminal pertenecerá en ese momento al programa que hemos mandado ejecutar, y cuando termine éste, se devolverá el control del terminal a la shell, volviendo a aparecer el prompt. Cuando queramos terminar la sesión de trabajo deberemos teclear el Control-D o exit, con lo cual la shell terminará y se volverá a pedir el nombre del siguiente usuario.



Actualmente existen dos familias de shell: la bash (Bourne Again Shell) y la tcsh, procedentes de otras

dos clases anteriores: la ksh (David Korn Shell) y la csh (Berkeley UNIX C shell). La primera es más usual para trabajos en general y la segunda está más orientada hacia la programación de guiones (shell scripts) (se verán posteriormente), aunque ambas tienen casi todas sus características comunes (salvo redirección, definición de varibales y startup) ya que proceden de la shell básica que es la sh, la shell de Bourne.

Lo que haremos en los siguientes temas es repasar estas características.

4.2. Comandos

Comandos Sencillos

Los comandos que puede ejecutar el sistema siempre tienen la misma estructura por un lado está el nombre del comando, a continuación las opciones que modifican la ejecución del mismo y que empiezan por el símbolo "-" (en los ejemplos siguientes de color verde), y por último los argumentos necesarios sobre los que toma o vuelca información, normalmente nombres de ficheros (en los ejemplos siguientes de color azul) .

Los comandos más utilizados en una sesión normal de trabajo son los que nos permitirán trabajar con ficheros, directorios y movernos en la estructura jerárquica que hemos visto anteriormente.

Los más importantes son los siguientes (sin indica sin opciones o sin argumentos):

- **Manejo de directorios**

```
pwd sin sin pinta directorio de trabajo
$ pwd
/usr/ramon/trabajos
$...
```

```
cd sin sin|directorio cambia de directorio (o sin, va la directorio
home)
$ cd ../textos
$...
```

```
mkdir sin directorio hace un nuevo directorio
$ mkdir /usr/ramon/graficos
$...
```

```
rmdir sin directorio remueve (borra) un directorio
$ rmdir graficos
$...
```

- **Manejo de ficheros**

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

```
$...
```

```
ls -a,c,l,r,s,... sin|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 son: a para dar todos los ficheros, c 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. En Linux incluso se pueden presentar los distintos tipos de ficheros por colores.

```
$ ll /usr/ramon
total 6
-rw-rw-rw- 1 grupo 7 alumnos 54 Nov 24 11:17 pepe.c
drw-rw-rw- 1 grupo 7 alumnos 654 Jul 01 13:11 ventas
-rw-rw-rw- 2 grupo 7 alumnos 3654 Nov 24 12:17 regis
```

donde en total nos dice el número de bloques de espacio listados (6), después en la siguiente línea y en el primer carácter nos dice el tipo de fichero que es, y a continuación los permisos del mismo, 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.

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

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

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 y si pulsamos q nos saldremos del programa (en algunos sistemas UNIX se utiliza el comando “pg” de similares características).

```
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 sin origen destino mueve ficheros a un directorio
f1 f2 fn directorio
$ mv registro almacen
$ mv reg1 reg2 reg3 muestras
$...
```

```
ln varias fichero directorio enlaza (link) un fichero a otro virtual
fichero fichero
$ ln /usr/ramon/ventas /usr/ramon/graficos
$...
```

En este comando la forma de enlazado se puede hacer de dosmaneras: hard, crea un enlace en el directorio sin crear el fichero enlazado (un mismo fichero con dos nombres, es decir, comparten el mismo inodo); soft, se crea un fichero que simplemente guarda información sobre como llegar al original. Los directorios sólo se pueden enlazar de esta manera.

```
rm sin 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 en UNIX no existe la orden undelete ni nada parecido y se se borrá no se podrá recuperar lo eliminado.

• Información

Hay un comando muy útil en caso de duda y es el que escribe el manual en la pantalla, su uso es simple:

```
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 que 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 comando `man` nos responderá con la primera información que encuentre por seccciones. El formato de salida será el del comando `more`, cuya actuación es parecida al comando `pg`. 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, como podrás comprobar si haces:

```
man -k apropos    o    man -K apropos
```

Las secciones del comando residen 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
10. ...

Por último y para acabar con este tema, podemos encadenar varios comandos en una sola línea de comandos simplemente separándoles con un **;**. También se pueden encerrar varios comandos entre paréntesis, en este caso hay que tener en cuenta que su ejecución se realizará dentro de una subshell, si se quisiera ejecutar la lista de comandos dentro de la misma shell se utilizarían las llaves con ; como terminación interior (es importante el espacio):

```
(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.
{ cd; pwd;} ; pwd    equivalente al anterior
```

Existen otras variantes de cadenas de ejecución, pero en este caso 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 status 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 de forma condicional, de tal manera que si se ha ejecutado el primero, 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)
```

4.3. Comodines

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 UNIX (en la shell) existen dos caracteres especiales el "*" que sirve para sustituir a una secuencia de 0 ó más caracteres y el "?" que sólo 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] sólo 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 caracter ^ (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`.

Otro caracter común a ambas shell es la tilde: ~ (126) -en el teclado de un PC se puede obtener con alt 126-. Este caracter sustituye al directorio home.

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 he intentamos acceder a él, piensa por ejemplo como podrías borrar un fichero que tiene en medio del nombre el caracter espacio. Al igual que en lenguaje C, también existen las secuencias de

escape y tienen aquí el mismo significado.

Consejo: puedes hacer pruebas creando ficheros con el comando `touch`.

4.4. Redireccion

Redirección

Otra característica importante de las shells del UNIX 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, de donde el programa toma los datos, por otro lado un canal de salida donde se dejan los resultados, y por último un canal de error que recibirá los mensajes de error que se produzcan durante la ejecución del proceso. Los dos primeros canales están asignados normalmente al terminal donde se trabaja (teclado y pantalla) y el tercero a la pantalla del terminal (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 pueden cambiar 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 junto con el comando, para ello utilizaremos la redirección de entrada:

```
$ programa < datos
```

donde datos es un fichero 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 y los saque en dos ficheros a la vez:

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

También podemos hacer que el fichero donde se sacan los datos no se inicialice cada vez que ejecutamos el programa si no que se añadan los nuevos datos a los antiguos, esto se consigue con ">>":

```
$ programa >> salida
```

Y como hemos dicho la equivalencia entre un fichero y un dispositivo en UNIX es total, por lo que podemos redirigir la entrada o la salida hacia un dispositivo:

```
$ programa > /dev/lp
```

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

```
$ cat pepe 2> juan
```

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.

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 subshell para corregirlo:

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

4.5. Canalización

Canalización

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 ésta. La canalización consiste en transmitir datos directamente de un proceso (comando) a otro, de tal manera que si un comando va a producir una salida que puede servir como entrada de otro comando, esto se pueda hacer directamente a través de la utilización del símbolo "|" (pipeline) que comunica dos comandos. Técnicamente hace una pipe (tubería) entre ellos y no a través de un fichero temporal y dos ejecuciones. Además no hay límite entre el número de procesos que se pueden encadenar:

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

Existe un comando especial `tee` que divide la salida de otro programa (su entrada) en dos, una de las cuales puede dirigirse hacia un fichero y otra encadenarse en la pipeline hacia otro comando:

```
$ programa1 | tee fichero | programa2
```

También podemos encadenar comandos a listas de comandos 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 fichero|write usuario&&echo funciono      Manda un fichero por pantalla a un
usuario y pone funciono sólo si realmente funcionó.
```

4.6. Background

Background

Hay dos formas de ejecutar un programa/comando una es la usual "foreground" y la otra es en background. En la primera el programa una vez ejecutado toma el control de la entrada/salida, en la segunda el control del terminal vuelve a la shell desde donde hemos ejecutado el comando (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`** (status de proceso) que no tiene argumentos y en el cual las opciones nos dicen que procesos representar (todos los del sistema, los de un usuario, etc.) y su formato. La forma más normal es:

```
$ ps
PID TT TIME COMMAND
3243 30 0:11 ksh
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 sólo lectura) hace de interfase (la veremos posteriormente en el capítulo de recursos) 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 necesaria para describir 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 (número) 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 comandos 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. 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. 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 una forma más sencilla de utilizarlos y no tener en cuenta el resto de procesos creados en el sistema. De esta manera nosotros podemos suspender un proceso que se esté ejecutando con la pulsación de `ctrl-z` (no confundir con ejecución en background, en ésta el proceso se está realmente ejecutando en otro plano) y al tener el control de la shell de nuevo, mandar un nuevo proceso a ejecutar. Para ver la lista de procesos de la sesión, jobs, tenemos en comando del mismo nombre `jobs`, que nos dará una lista de los mismos entre corchetes:

```
$ jobs
```

```
[1]+ Running ls --color=tty -lR / &
```

Para ejecutarles sólo tenemos que utilizar los comandos fg y bg, que ponen respectivamente al job en ejecución *foreground* o *background*.

También existen comandos específicos para ejecutar en background un proceso o bien a una determinada hora o bien cuando el sistema tenga tiempo de CPU libre, estos comandos son **`at`** y **`batch`** (primitivo sistema de colas) de los que se puede encontrar más información en el manual. Siempre se puede ver la cola de procesos ejecutados con at con **atq** y también se pueden borrar con **atrm**. No todos los usuarios tienen el privilegio de poder ejecutar programas con at.

Por último en algunas ocasiones nos interesa mantener un trabajo después de acabar la sesión de trabajo (obviamente no puede tener ni entradas ni salidas), esto normalmente no es posible ya que la shell manda la señal de SIGHUP para terminar con todos los procesos. Esto se puede evitar si ejecutamos un comando antecediéndole `nohup`:

```
$ nohup ls -R /
```

4.7. Variables

Variables de shell

Otra característica potente de las shell es la definición de variables que servirá, primero para guardar datos y después para modificar su comportamiento. Esto se hace o bien a través de la creación de variables de shell del 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 UNIX es distinto una letra mayúscula que una minúscula y que las variables se suelen poner en mayúsculas. Una vez definida la variable, se podrá utilizar en cualquier comando del sistema simplemente anteponiendo a su nombre el carácter “$”. Por ejemplo:

```
$ DIRECTORIO=/home/pepe/proyecto/sesion/trabajo/grupo     Definición
$ echo $DIRECTORIO                                        Uso
```

En cuanto a las variables definidas por la shell, las más usadas son (hacer `man bash` para ver todas):

`PATH` Donde se guardan los directorios donde se pueden encontrar ficheros ejecutables.
`HOME` E l directorio de trabajo (home) del usuario.
`PS1` El string que se utiliza como prompt ($).
`PWD` El directorio donde estamos.

En el primer caso la definición de la variables será el conjunto de directorios donde la shell va a buscar ejecutables (en otros directorios no se podrá ejecutar nada) separados por ":" . Si hacemos `echo $PATH` se podría obtener algo parecido a :

```
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:$HOME/bin
```

La segunda variable se puede usar como hemos visto para obtener el directotio home de un usuario. En el tercer caso se puede usar como valor un string y alguno de los modificadores que existen (hacer `man bash` para ver todos). Los modificadores más importantes 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]\$.

Debemos tener en cuenta que la definición de una variable sólo tiene efecto en esa sesión de shell, si queremos que sirva para posteriores sesiones (variables de entorno o *environment variables*) hay que utilizar el comando `export` (realmente es un comando interno [*built-in*] de la shell). Esto es importante en el siguiente apartado en que veremos que es una macro, ya que las macros se ejecutan en una subshell. El entorno definido (conjunto de variables exportadas) se pasa también a cualquier programa ejecutado desde la shell en el tercer argumento de *main* (en el caso de lenguaje C).

Otro comando interesante es el `set` nos pondrá en pantalla las variables que tenemos asignadas (si sólo queremos conocer una podremos hacer `echo $variable`). Su contrapartida está en `unset` que borra una variable. Servirán tanto para la shell `bash` como para la `tcsh`, en el caso de esta última, que veremos a continuación, también existe el comando `setenv` para ver las variables de entorno.

La shell tcsh

En las shell derivadas de la c, como la tcsh, se utiliza este mismo comando `set` para definir las variables (con la opción -r se definen constantes y no variables). La forma usual de hacerlo es con el nombre de la variable seguida de "=" y del valor (si es una sola palabra). Si el valor fuera de varias palabras se utilizan los paréntesis y para el *prompt* el apóstrofe:

```
$ set variable=valor
$ set path=($path . $/home/bin)
$ set path=(/usr/local/bin /bin /usr/bin .)
$ set prompt='%/ %M %n %# hola '
```

El último pondrá el directorio, el host, el usuario, el >, y el string hola. Más modificadores de prompt se pueden ver al hacer `man tcsh`.

Si al usar el set el `valor` es nulo servirá para borrarla, en ambas shell se puede usar el comando `unset` para borrar una variable. Como en el caso de la bash (sh), sólo valen para una única sesión, si queremos exportarlas tendremos que usar en este caso el comando `setenv` en vez de `set`, ya que `export` no existe.

4.8. Macros

Shell script (macros)

Un aspecto importante de las shell de UNIX y esencial en la administración de sistemas, 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* (guión) 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 sólo nombre.

Básicamente, un macrocomando es un fichero de texto que tiene el atributo de ejecutable (o puede ser ejecutado con `sh macro`) y en el que hemos escrito una serie de comandos que queremos ejecutar secuencialmente, incluyendo redirección, canalización y background. Pero la estructura de los macrocomandos puede llegar a la misma complejidad que la de un programa de C, de hecho dentro de un macrocomando se pueden utilizar las mismas estructuras de control 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 minishell (bourne) a la que le damos el nombre del script:

```
sh macro
```

El comando `sh` nos creará otra shell que ejecutará la macro de tal manera que cuando termine ésta, también terminará la nueva sub shell. Otra forma de ejecutarla es cambiando el modo del fichero haciéndolo ejecutable como se verá en el siguiente capítulo.

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 tomar con el comando interno `shift`.

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

`$1 a 9` Variables de parámetros posicionales.
`$0` El nombre de la propia macro.
`$#` El número de argumentos dados.
`$?` El código exit del último comando ejecutado, cuando se ha ejecutado sin errores valdrá 0.
`$!` El identificador de PID del último proceso ejecutado.
`$$` El identificador del proceso.
`$*` Un string conteniendo todos los parámetros pasados empezando en $1.

Esto se ha utilizado algunas veces para redefinir comandos de otros sistemas operativos como comandos de UNIX. Por ejemplo, podríamos hacernos nuestro propio `dir` del MS-DOS escribiendo un fichero de texto que se llamara dir, que se ejecutara como `dir directorio` y que contuviera :

```
ls -l $1
```

En este caso $0 sería el propio nombre de la macro:`dir`, y $1 el primer argumento: `directorio`. De todos modos en la práctica no se usan las macros para este menester, ya que tenemos el comando interno `alias`, que permite realizarlo más simplemente:

```
$ alias cd.. = "cd .."
$ alias dir = "ls -l"
```

en estos dos ejemplos hemos redefinido el comando `ls` como `dir` y hemos evitado un error frecuente que se comete al no dejar espacio entre `cd` y el directorio "..". Si quisieramos eliminarlos utilizaríamos el comando interno de la shell `unalias`.

Las variables (información del exterior) también se pueden leer del 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). El siguiente

ejemplo nos demuestra como 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"
```

El comando `echo` (el mismo que el del tema anterior) 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.

Si un comando necesita en una macro unas cuantas líneas de texto se utilizan los *documentos here* que comienzan por `<<- identificador` y acaban en `identificador`, para más detalles `man bash`.

En algunos casos se suele utilizar una línea de comienzo especial que indica con que programa se tiene que ejecutar la macro, así, podríamos haber comenzado el *script* con la línea:

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

que es la shell por defecto. Como prueba cambia este comando por el `cat` y mira que ocurre.

Lo que hemos visto hasta ahora haría las veces (en un programa) de declaración de varibales y entrada/salida, pero como hemos dicho, en una macro se pueden utilizar instrucciones de control de flujo como selección e iteración. 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 este caso 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 puede encontrarse con `man test`), precisamente las opciones nos dirán lo que tenemos que comprobar (test). En lo que se refiere a ficheros como argumentos, las principales opciones son: -e para existencia, -d para directorio y -f para fichero regular.

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

En la mayoría de las shell modernas, en una macro en vez de utilizar test se puede usar `[ ];` que es equivalente: `[ -e fichero];`. También admite argumentos de tipo expresión lógica realizando entre ellos operaciones lógicas: AND (-a) y OR (-o), string y enteros sobre los que puede establecer comparaciones. 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 son:

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

Las expresiones construidas con test (o la ejecución de cualquier comando que como hemos visto también es una expresión lógica) son utilizadas para cambiar el flujo de control de la macro, la orden más básica es:

```
if expresion
     then ordenes
fi
```

Un ejemplo de utilización sería:

```
if test ! -f $FICHERO
then
     if test "$MENSAJE" = "si"
     then
         echo "$FICHERO no existe"
     fi
fi
```

En este caso estamos usando el comando `test` combinado con la sentencia `if` (terminado con `fi`). Como hemos dicho, al igual que utilizamos estas sentencias en C también las podemos utilizar en las macros, en el ejemplo anterior miramos si el fichero dado por la variable FICHERO existe y es regular, si no existe comprobamos si la variable MENSAJE está puesta a si entonces pintaremos un mensaje (observar que hemos utilizado el operador = del comando `test`).

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

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

```
if who | grep –s pepe > /dev/null
then
    echo pepe esta en el sistema
else
    echo no esta
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 opción `–s` indica que no se presenten mensajes de error y la salida estándar se desvía a nada 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
```

Igual que existen sentencias condicionales, también existen sentencias iterativas, la primera de ellas es la sentencia `for` que utiliza una variable de indice 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:

```
for var in lista de palabras
do
     comandos
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 $*
do
     if who | grep $i > /dev/null
     then
          echo $i esta en el sistema
     else
          echo $i no esta
     fi
done
```

Otros tipos de bucle son el `while` y el `until` cuyo significado es claro y cuya estructura aparece a

continuación.

```
while lista de comandos
do
    comandos
done
```

```
until lista de comandos
do
    comandos
done
```

En el siguiente ejemplo podemos esperar a que un usuario salga del sistema (sleep 60 parará la ejecución durante un minuto):

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

Arranque (start up) de una shell

Cuando hacemos definiciones de variables o cambios de las existentes y queremos que siempre estén presentes en cualquier sesión de trabajo que establezcamos, deberemos definirlas y exportarlas en ficheros especiales (están ocultos ya que empiezan por ".") de definición de shell. Estos ficheros no son más que macros que se ejecutan en el inicio de la sesión y que por lo tanto también pueden ejecutar comandos como por ejemplo alias. La única dificultad es que cada tipo de shell (incluidos los entornos gráficos) tiene sus propios ficheros de arranque. Los que nosotros tenemos presentes están en el directorio de trabajo: $HOME y se pueden ver con:

```
ls –a $HOME
```

Una de las macros típica es `.profile` , que en el caso de la shell bash se llama `.bash_profile`. Cada vez que abramos una sesión de trabajo, la shell ejecutará este fichero y después nos devolverá el prompt. En este fichero se suelen poner configuraciones del terminal que estamos usando, por ejemplo: stty erase ^h o definiciones de variables de la shell como PS1 o PATH.

En el caso de la tcsh habrá que colocar la definición en el fichero `.cshrc (.tcshrc)` o `.login`. La diferencia entre ellos es que el fichero `.login` no es ejecutado por la shell cuando no se produce un procesos de login, por ejemplo con el comando `su`.

Otras programas de creación de "macros"

Cuando el trabajo que debe realizar la macro (script) se hace más complejo, las herramientas disponibles en la shell se quedan cortas, por eso existen comandos avanzados como el awk (de los apellidos de los autores Aho, Weinberger y Kernighan -gawk en su versión GNU-) que básicamente realizan búsquedas en ficheros y procesan patrones a través de expresiones regulares. Las expresiones regulares se utilizan en más programas como el editor vi (ver apartado 5.2), el comando grep (apartado 5.3) o incluso otros intérpretes de comandos como el sed. Su uso sería:

```
awk programa fichero
```

donde programa contendría las expresiones regulares y patrones a buscar en `fichero`. Queda fuera de una asignatura básica explicar su funcionamiento (aunque no es muy complejo) pero como muestra valga un botón, supongamos que tenemos un fichero organizado en 6 columnas donde la primera columna es un producto, la segunda su precio, y de la tercera a la sexta las ventas trimestrales. Quremoa añadir al fichero dos columnas más con las ventas totales del año y el importe. En este caso programa tendría este aspecto:

```
{total=$3+$4+$5+$6; print $0, total, total*$2}
```

que se ejecutará en dos etapas (;) en la primera se calcula el total y en la segunda se pone la línea original ($0) seguida de las dos nuevas columnas.

También han surgido nuevos lenguajes interpretados más potentes que estos comandos entre los que cabe destacar perl (Practical Extraction and Report Language -http://www.perl.org/-) y phyton (http://www.python.org/), más sencillo y claro que el anterior y a caballo entre un lenguaje interpretado tipo perl y un lenguaje de medio nivel como C o Java. El que os escribe es un odiador nato de perl a pesar de que reconozco que es muy potente, pero como dicen del UNIX, si un chimpancé se pusiera delante de un teclado lo primero que escribiría sería una línea de Perl.

4.9. Edición

Edición de Línea

La principal ventaja de las nuevas shell frente a la clásica sh (Bourne) es fundamentalmente la posibilidad de editar la línea de comandos. Es decir, tener almacenadas todas las órdenes escritas (*history*) y poderlas recuperar y editar fácilmente (en la `bash` se pueden ver con el comando interno `history`, y hay dos variables que controlan por un lado el número de órdenes y por otro lado donde: HISTSIZE y HISTFILE). En el caso de un PC, las líneas de comando anteriormente ejecutadas pueden ser editadas a través de las teclas de flechas del teclado. Normalmente también se sigue una nomenclatura de desplazamiento similar al editor de textos que exista definido para el usuario, normalmente el `vi` o el `emacs`:

```
^F mueve al siguiente carácter        ^A mueve al comienzo
de línea
^B mueve al carácter anterior         ^E mueve al final de
línea
^D   borra   carácter                       Delete   borra
carácter a la izquierda
^K borra hasta el final de la línea   ^Y undelete
```

También la shell puede completar la línea de edición actual utilizando la tecla `tabulador`, para ello se tiene en cuenta la fracción de la línea editada. Así, si la pulsamos al teclear un comando la shell nos lo completará con el comando que empiece con lo que hasta el momento hemos pulsado, o si estamos utilizando el nombre de un fichero o directorio, con su nombre. Si existe algún tipo de ambigüedad, la shell pitará y presentará en pantalla las posibilidades, a lo cual el usuario responderá con la pulsación de algún carácter que resuelva la ambigüedad.

Otra posibilidad muy útil es la expansión del *history*, de tal manera que si queremos repetir y/o modificar alguna orden introducida, teclearemos el expansor "!" seguido del comienzo de la orden, la shell nos pondrá automáticamente (lo último en el `history` que coincida) lo que va a ejecutar y después de un lapso breve lo ejecutará. Así,

```
$!man b
```

ejecutará `man bash` si es lo último que empieza por man b.

En el caso de la `tcsh`, el comportamiento puede ser modificado a través de los valores de las variables de la shell relacionadas: `autolist` y `fignore`. Con la primera nos pondrá la lista de posibilidades automáticamente, con la segunda se pueden ignorar algunas terminaciones de ficheros en esa lista. También a través de la variable `correct` se puede activar el servicio de autocorrección para que la shell compruebe la corrección del comando editado automáticamente. Si hay algún error, la shell nos propondrá la versión correcta, a lo cual nosotros responderemos con y, n, e o a, para decir si estamos o

no de acuerdo, para dejar la que estaba, o para abortar. El conjunto de variables, así como cualquier ampliación, se puede ver con `man tcsh`.