**Práctica 1 Introducción a Linux (Cygwin)**

# Objetivos

El objetivo principal que se persigue en esta práctica es que el alumno se familiarice con el entorno de trabajo en el que se desarrollarán las prácticas posteriores: Linux.

En esta primera parte aprenderá a usar el sistema operativo Linux. Gran parte de los conceptos no son específicos de Linux sino que pueden aplicarse igualmente a cualquier sistema operativo de tipo UNIX. Además se aprenderá a controlar los procesos que ha lanzado su usuario y a modificar los permisos de sus ficheros.

# Primeros pasos

Para esto utilizaremos Cygwin un emulador de Linux para Windows, pero funcional en la mayoría de los casos. Al hablar de UNIX, nos referiremos a Linux o a Cygwin en este caso.

# Manejando ficheros

Los sistemas UNIX al ser multi-usuario requieren que cada cuenta de usuario pueda tener diferente acceso a los directorios y ficheros. Nada más entrar la *shell* está esperando órdenes y está posicionada en el directorio *home* del usuario. Observa cuál es tu directorio *home* con el comando

$ pwd

Puedes crear un fichero con cualquiera de los siguientes comandos:

$ echo “hola” > fichero1

$ cat > unfichero

escriba algo

^D (pulsar control+D)

$ touch fichero\_tres

Usa el comando ls para comprobar que se han creado y los comandos siguientes para ver su contenido:

$ cat unfichero

$ more fichero1

$ less unfichero (para salir pulsa q)

El sistema de ficheros soporta una serie de permisos que dicen quién tiene derecho a usar o no cada fichero. Puedes ver estos permisos con el comando ls. Puedes consultar todas las opciones de un comando mediante el sistema de ayuda de UNIX proporcionado por el comando man, por ejemplo man ls.

Haz un listado de los ficheros de tu directorio en formato largo y observarás algo parecido a ésto:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| $ ls ­l |  | | | | | | | |
| drwxr­xr­x | 3 | usuario | staff | 4096 | Dec | 16 | 08:14 | Desktop |
| ­rw­r­­r­­ | 1 | usuario | staff | 41685513 | Sep | 27 | 13:57 | usuario.tgz |
| ­rw­r­­r­­ | 1 | usuario | staff | 1000000 | Nov | 12 | 20:00 | oo |
| ­rw­r­­r­­ | 1 | usuario | staff | 1000000 | Nov | 12 | 20:02 | ooo |
| drwxr­xr­x | 2 | usuario | staff | 4096 | Oct | 6 | 13:07 | prueba\_c |
| ­rwxr­xr­x | 1 | usuario | staff | 94 | Nov | 25 | 11:56 | vnc\_to\_usuario.bash |
| drwx­­­­­­ | 10 | usuario | staff | 4096 | Oct | 6 | 11:26 | W2000 |

El primer bloque de letras y guiones en cada línea indica los permisos sobre el fichero. La primera letra indica el tipo de fichero. Las 9 restantes se pueden considerar en grupos de 3. Cada grupo indica los permisos que tienen diferentes tipos de usuarios.

El sistema de ficheros nos permite organizar los ficheros en directorios y subdirectorios. Prueba el funcionamiento de los siguientes comandos

$ pwd # muestra el directorio actual

$ mkdir midir # crea un subdirectorio midir en el directorio actual

$ ls # para comprobar que hay un directorio nuevo

$ cd midir # el directorio actual pasa a ser midir

$ pwd # comprobación de lo anterior

$ cd .. # el directorio actual

$ pwd # comprobación

Los ficheros y directorios pueden referenciarse con paths absolutos o relativos al directorio actual. El directorio raíz es /

Prueba estos comandos:

$ more /etc/services

$ cd /etc

$ more services

$ more ~/unfichero # ~ es el directorio home de ese usuario

$ cd # o cd ~

$ pwd

$ cd .. # .. es el directorio padre del directorio actual

$ pwd

$ cd ../../../etc

$ more ./services # . es el directorio actual

Comprueba el efecto del permiso de ejecución (x) para un directorio:

$ cd

$ mkdir midir/cosas

$ cp /etc/services midir/cosas/servicios

$ chmod u­x midir

¿Puedes acceder al fichero midir/cosas/servicios? ¿Puedes entrar en el directorio cosas?

¿Por qué?

**Más sobre la *shell***

Como ya se ha comentado, el programa encargado de leer lo que escribe el usuario y lanzar los programas que pide se llama genericamente *shell*. En UNIX se han escrito distintas *shells* aunque todas tienen las mismas funciones básicas. Si abres un terminal Linux lanzará una *shell* para atender sus órdenes. La *shell* que se está usando es en realidad un programa llamado bash. Puedes observarlo usando el comando ps que muestra los procesos que están ejecutando en nuestro ordenador. Prueba ésto:

$ ps

PID TTY TIME CMD

11486 pts/3 00:00:00 bash

11585 pts/3 00:00:00 ps

$

El comando ps sin opciones muestra sólo los procesos asociados a la misma terminal. Como puede ver al lanzar ps, vemos dos procesos, uno es el proceso ps propiamente dicho, el otro es el proceso que está corriendo bash.

bash no es más que un programa. Para comprobarlo intenta lanzarlo. Algo así:

$ bash

$

¿Qué es lo que ha pasado? ¿Nada? Utiliza el comando ps para comprobar que en realidad ahora hay dos *shells* corriendo. Básicamente bash estaba esperando comandos y se le ha ordenado que lance un segundo bash y lo ha puesto a funcionar. El nuevo bash ha tomado el control y espera órdenes mientras que el primer bash está esperando a que acabe el segundo como haría con cualquier comando.

¿Cómo puedes cerrar el segundo bash y volver al primero? A estas alturas debes de conocer al menos dos formas...

Puedes probar también otras *shells* diferentes, los tradicionales se llaman sh, ksh, csh, tcsh y bash. Los dos últimos que son los más modernos e incluyen todo lo que incluían los demás (tcs es una evolución de csh y bash es una evolución de sh/ksh así que hay más bien dos familias). Verás que tu Linux no lleva ksh.

Como recordarás la función principal del *shell* permitir lanzar programas (comandos) y comunicarse con ellos. En UNIX los programas tienen una entrada estándar de la que reciben datos llamada *stdin* y una salida estándar llamada *stdout* en la que se representan los datos que generan. (Para los usuarios avanzados diremos que también hay una salida de errores diferente de la salida estándar pero eso dejaremos que lo aprendas por su cuenta cuando conozcas más UNIX)

Por defecto la entrada estándar de la *shell* y de cualquier proceso que se lance desde ella es el teclado y la salida estándar será la pantalla. Pero gran parte de la flexibilidad de uso de programas desde la *shell* viene de que podemos cambiar o redirigir estas entradas y salidas.

Para redirigir la salida de un comando hacia un fichero podemos usar el símbolo >

seguido del nombre del fichero detrás de un comando. Por ejemplo

$ date

$ date > fichero1

$ more fichero1

$ cal > fichero1

$ more fichero1

También podemos redirigir la salida con el símbolo >> lo que tiene un efecto un poco diferente. Descúbre tú mismo...

$ date >> fichero1

$ more fichero1

$ cal >> fichero1

$ more fichero1

La entrada se redirige con el símbolo <. Por ejemplo puedes contar las palabras o las líneas de un fichero lanzando el comando wc

$ wc ­w < fichero1

$ wc ­l < fichero1

Prueba también el comando grep que permite buscar líneas que contengan una cadena determinada:

$ grep linea

Este programa busca líneas que contengan la cadena linea observa que repite automáticamente las líneas en las que encuentra algo. Consulta la ayuda de este comando ejecutando man grep.

Prueba a usar el comando grep para encontrar todas las líneas del fichero

/etc/services que contengan el texto "80" utilizando la redirección de entrada.

La tercera forma de redirigir la salida es encadenar la salida de un programa con la entrada del siguiente. Esto se conoce en UNIX como usar una *pipe* (tubería) y se hace con el símbolo |

Como veremos más adelante esto permite encadenar programas que realizan procesados simples sobre ficheros para lograr una tarea mayor combinando procesados. De momento veamos qué se puede hacer con los comandos que conoces...

$ date | wc ­c

$ ls ­l | wc ­l

$ cat /etc/services | grep web | wc ­l

¿Qué hace el último de los comandos anteriores?

Prueba también a redirigir a un visor de pantalla completa como el comando more o

less

$ cat /etc/services | more

$ cat /etc/services | less

También se puede usar una combinación de redirecciones. Por ejemplo podemos usar el comando cat para copiar un fichero redireccionando la entrada y la salida. El comando cat en realidad lo único que hace es leer su entrada estándar y copiarla a su salida estándar. ¿Cómo copiarías un fichero con éso?

Otra utilidad de la *shell* es que permite especificar nombres de ficheros que cumplan un patrón determinado mediante el uso de comodines. Los comodines nos permiten especificar una cadena que se comparará con los nombres de ficheros y se sustituirá por todos los nombres que coincidan. Así podemos usar el comando:

$ ls /etc/s\*

Para listar todos los nombres de ficheros que empiezan por /etc/s. Los comodines que se permiten son los siguientes:

\* cualquier cadena, incluyendo la cadena vacía

? cualquier caracter

[...] cualquiera de los caracteres entre [ y ]

Con un solo comando, lista los ficheros del directorio /usr/lib que empiecen por lib, liste también los ficheros de /etc que terminen en .conf.

# Controlando los procesos

La multitarea es una de las funcionalidades más importantes de UNIX. Todo programa que lanzamos en UNIX genera un proceso. El proceso es el programa en ejecución. El sistema operativo tiene una tabla de procesos e identifica a cada proceso con un identificador de proceso, PID, que es un número entero. Como ya hemos visto el comando ps nos deja ver los procesos. Hasta ahora sólo lo habíamos usado para ver los procesos asociados a un terminal, pero con opciones podemos ver más cosas.

Dado que al comando sólo se le pasan opciones, ha evolucionado hasta no utilizar el ­ delante de ellas. Estas son las combinaciones más útiles. Averigua qué hacen:

$ ps

$ ps ax

$ ps axu

$ ps axw

$ ps axl

$ ps axj

Los procesos en UNIX son lanzados por otros procesos. Se dice que un proceso se bifurca (*fork*) y crea un proceso hijo. Por tanto los procesos se agrupan en una jerarquía de descendientes. Por ejemplo todos los procesos que lance desde su terminal serán hijos del proceso *shell* que está usando.

$ ps ef

Al menos recuerda una forma de listar todos los procesos de la máquina. Para ello practica un poco siguiendo los pasos que vienen a continuación:

Usa el comando sleep para crear un proceso simple:

$ sleep *numero\_de\_segundos*

Este comando espera el número de segundos indicado y acaba. Prueba a lanzar un sleep 600 en un terminal. Mientras se está ejecutando abre un nuevo terminal y pide la tabla de procesos y prueba a buscarlo con:

$ ps

$ ps axu

¿Cómo podrías hacer un comando que muestre una línea por cada proceso sleep que está corriendo en la máquina?

Como has visto, un proceso corriendo no devuelve el control a la *shell* hasta que termina, de forma que un proceso largo que se lanza en una terminal la bloquea hasta que acabe. Esto no supone mucho problema si estamos en el modo gráfico ya que podemos lanzar más terminales. En cualquier caso, un proceso que está corriendo en la terminal se dice que está corriendo en primer plano (*foreground*). No todos los procesos que están corriendo en un UNIX han sido lanzados por un terminal. Un proceso corriendo de fondo (*background*) funciona independientemente de las acciones del usuario. Se entiende que debe estar preparado para no necesitar datos del teclado ni esperar imprimir en la terminal. Para lanzar un proceso que no bloquee el terminal debes añadir un & al final del comando.

Pruebe con este ejemplo:

$ ( sleep 30 ; echo "fin" )

Los paréntesis nos permiten agrupar varios procesos que se lanzan en una *shell* aparte como si fuera un comando. Mira con ps si quieres saber lo que está pasando. El comando esperará 30 segundos y escribirá fin. Puedes parar el programa con Control+c (a veces lo verás escrito como ^C).

Prueba a lanzar el ejemplo con un & para que se ejecute en segundo plano:

$ ( sleep 30 ; echo "fin" ) & [1] 2325

$ ps

PID TTY TIME CMD

2314 pts/0 00:00:00 bash

2325 pts/0 00:00:00 bash <<<< se ha lanzado un bash para hacer el parentesis 2326 pts/0 00:00:00 sleep << el bash a su vez ha lanzado un proceso sleep 2327 pts/0 00:00:00 ps

Verás que vuelves a tener el control del terminal y se indica el número de proceso que se le ha dado al comando. Obsérvalo con el comando ps.

Los procesos pueden recibir señales mediante el comando kill y permiten cierto control sobre ellos. Se llama así porque la señal que envía por defecto causa (normalmente) que el proceso que la recibe termine y sea eliminado de la tabla de procesos. La sintaxis básica es

$ kill ­*nombre\_de\_la\_señal PID*

$ kill ­*numero\_de\_la\_señal PID*

Y puede interpretarse como: manda la señal indicada (bien con nombre\_de\_la\_señal o con su equivalente numero\_de\_la\_señal) al proceso con el identificador PID. Hay varias señales en UNIX y éstos son los nombres y números de las más importantes (puedes consultar el resto en el man):

|  |  |  |
| --- | --- | --- |
| 1 | HUP | Hung Up (literalmente colgar (el teléfono)) |
| 2 | INT | Interrumpir programa |
| 9 | KILL | Parada del programa no interrumpible |

La señal HUP (1) era originalmente para avisar de que el módem ha colgado la llamada al que está asociado el proceso, dependiendo del programa su reacción puede ser terminar o en la mayoría de los servicios de red lo que hacen es reiniciarse y volver a leer el fichero de configuración.

La señal INT (2) está pensada para interrumpir el programa. La mayoría de programas la interpretan como señal de que el programa debe acabar. Algunos programas pueden interceptarla y entender otra cosa (por ejemplo emacs lo hace). Esta es la señal que se manda a un proceso cuando pulsamos ^C en la consola.

La señal KILL (9) es una señal de que el programa va a ser eliminado de la tabla de procesos. El programa para y se elimina sin poder interceptar la señal.

Así que normalmente se utiliza

$ kill *pid*

para matar un proceso y $ kill ­9 *pid*

para matar un proceso que se resiste al kill normal

Un usuario sólo puede mandar señales a sus procesos y sólo el superusuario (root) es capaz de mandar señales a cualquier proceso de cualquier usuario.

También puedes utilizar el comando killall que a veces es más cómodo. Pero killall es también más peligroso, ¿por qué?