Introducción a Sistemas Operativos: Usando el shell

Clips xxx
Francisco J Ballesteros

1. Comandos como herramientas

En UNIX la idea es combinar programas ya existentes para conseguir hacer el trabajo. Actualmente, por desgracia, hay muchos usuarios de UNIX que lo han olvidado y corren a implementar programas en C u otros lenguajes para tareas que ya se pueden resolver con los programas existentes.

Por ejemplo, quizá conozcas un programa llamado *head(1)* que imprime las primeras líneas de un fichero. Si recuerdas que *seq(1)* escribe líneas con enteros hasta el valor indicado, verás que en

```
unix$ seq 10 | head -2
1
2
unix$
```

`head` ha impreso las primeras dos líneas aunque `seq` escriba 10 en este caso. Pues resulta que no era preciso programar `head`, dado que con un editor de *streams* (texto según fluye por un pipe) ya era posible imprimir las primeras *n* líneas. Baste un ejemplo:

```
unix$ seq 10 | sed 2q
1
2
unix$
```

Se le pide educadamente a *sed(1)* que termine tras imprimir las primeras dos líneas sin hacer ninguna edición en ellas y ya lo tenemos.

Saber utilizar y combinar los comandos disponibles en UNIX te resultará realmente útil y ahorrará gran cantidad de tiempo. Además, dado que estos programas ya están probados y depurados evitarás bugs innecesarios. Simplemente utiliza el manual (¡Y aprende a buscar en él!) para localizar comandos que hagan o todo o parte del trabajo que deseas hacer. Combina varios de ellos cuando no sea posible efectuar el trabajo on uno sólo de ellos, y recurre a programar una nueva herramienta sólo como último recurso. Los comandos de unix son una caja herramientas, ¡úsala!.

La utilidad del shell es, además de permitirnos escribir comandos simples, permitirnos combinar programas ya existentes para hacer otros nuevos. Ya lo hemos estado haciendo anteriormente, pero ahora vamos a dedicarle algo de tiempo a ver cómo programar utilizando el shell. Recuerda que *sh(1)* está disponible en todos los sistemas UNIX, por lo que aprender a utilizarlo te permitirá poder combinar comandos y programar scripts en todos ellos.

2. Convenios

Para facilitarte el trabajo a la hora de utilizar el shell, deberías ser sistemático y seguir tus propios convenios. Por ejemplo, si las funciones en C las escribimos siempre como en

```
int
myfunc(int arg)
{
    ...
}
```

entonces sabemos que todas las líneas con un identificador a principio de linea seguido de un "`(`" está definiendo una función.

El comando *egrep(1)* imprime líneas que encajan con una expresión (como veremos más adelante). Si hemos sido sistemáticos con nuestro convenio para programar funciones, podríamos pedirle que escriba todas las líneas que declaran funciones en un directorio dado. No te preocupes por cómo se usa `egrep` por el momento, presta atención a cómo un simple convenio nos permite trabajar de forma efectiva.

Podemos utilizar el comando *egrep(1)* para ver qué funciones definimos y en qué ficheros y líneas:

```
unix$ egrep -n '^[a-z0-9A-Z_]+\(' *.c
alrm.c:11:tout(int no)
alrm.c:17:main(int argc, char* argv[])
broke.c:8:main(int argc, char* argv[])
fifo.c:14:main(int argc, char* argv[])
...
```

O para contar cuántas funciones definimos...

```
unix$ egrep -n '^[a-z0-9A-Z_]+\(' *.c | wc -l
      24
unix$
```

Programando en un entorno integrado de desarrollo (IDE o *integrated development environment*) puedes hacer este tipo de cosas pulsando botones con el ratón. ¡Pero ay de ti si quieres hacer algo que no esté disponible en tu IDE! Con frecuencia notarás que estás haciendo trabajo repetitivo y mecánico, ¡incluso si utilizas un IDE en lugar de un editor sencillo! Cuando te suceda eso, recuerda que seguramente puedas programar un script que haga el trabajo por ti.

El shell es muy bueno manipulando ficheros, bytes que proceden de la salida de un comando y texto en general. Igual sucede con los comandos que manipulan ficheros de texto en UNIX. Para programar utilizando el shell hay que pensar que no estamos utilizando C e intentar escribir comandos que operen sobre todo un fichero a la vez o sobre todo un flujo de bytes a la vez. La idea es ir adaptando los datos que tenemos a los que queremos tener. Otra estrategia es adaptar los datos que tenemos para convertirlos en comandos que, una vez ejecutados, produzcan el efecto que deseamos. Iremos viendo esto poco a poco con los ejemplos que siguen.

En muchos casos es posible que estés intentando resolver problemas que has creado tu mismo. Lo mejor en la mayoría de los casos suele ser intentar no crear los problemas en lugar de resolverlos. Baste un ejemplo.

Supón que estás programando una aplicación que requiere de un fichero de configuración. Tradicionalmente estos ficheros suelen guardarse en el directorio `$HOME` con nombres que comienzan por "`.`", pero eso nos da igual en este punto. Ya sea por seguir la moda o por no saber cómo leer líneas y cómo partirlas en palabras, quizá programemos la aplicación utilizando XML como formato para el fichero de configuración. ¡Qué gran error! Al menos, en los casos en que nuestro fichero de configuración necesite información tan simple como

```
shell /bin/sh
libdir /usr/lib
```

para indicar qué shell queremos que ejecute y qué directorio queremos que utilice para almacenar sus

librerías (por ejemplo). El lugar de un fichero tan sencillo como el que hemos mostrado, quizá termines con algo como

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE MyConfig SYSTEM "Crap_Config.dtd">
<Crap_Config>
    <Parameters>
        <Param>
            <Name>shell</Name>
            <Value>/bin/sh</Value>
        </Param
        <Param>
            <Name>libdir</Name>
            <Value>/usr/lib</Value>
        </Param
    </Parameters>
</Crap_Config>
```

Es cierto que tienes librerías ya programadas para la mayoría de lenguajes que saben entender ficheros en este formato, leerlos y escribirlos. Pero eso no es una excusa si el fichero en cuestión guarda información que puedas tabular (líneas de texto con campos en cada línea). En realidad, ni siquiera es una excusa si necesitas guardar una estructura de tipo árbol que sea sencilla, como por ejemplo este fichero:

```
/
    home
        nemo
    bin
    tmp
```

que podría indicar con qué partes de un árbol de ficheros queremos trabajar.

Pero volvamos a nuestro fichero de configuración. Supón que deseas comprobar que para todos los usuarios que utilizan tu aplicación en el sistema, los shells que han indicado existen y son ejecutables y los directorios existen. ¿Cómo lo harás? Con el formato sencillo para el fichero bastaría hacer algo como

```
unix$ shells=‘egrep ’^shell’ /home/*/.myapp | cut -f2‘
```

para tener en `$shells` la lista de shells y luego podemos utilizar esta variable y un bucle `for` del shell para comprobar que existen. Si has utilizado XML también podrías programar un script para hacerlo, pero te resultará mucho más difícil. Te has buscado un problema que no tenías.

Pero vamos a ver cómo usar el shell para hacer este tipo de cosas...