

Práctica 4 - Programación de comandos

1. Introducción

En prácticas anteriores se utilizaron varios comandos básicos de UNIX como `ls`, `cd`, `grep`, y `chmod`. A lo largo de esta práctica se explicará qué son realmente los comandos y se pedirá implementar en Java comandos propios. El objetivo es que se comprenda cómo los sistemas UNIX permiten el uso de los distintos comandos habituales, que se vea que es relativamente sencillo crear comandos propios con el funcionamiento que se requiera y practicar el manejo del compilador de Java y la programación con este lenguaje.

Esta práctica puede hacerse fuera del laboratorio en cualquier sistema operativo UNIX donde instalemos Java, en los escritorios virtualizados de la universidad¹, en los PC SC de los armarios del laboratorio o en los PCs del otro lado del aula.

¹ <https://vdibroker.unavarra.es/>

2. Los comandos en UNIX

La mayoría de los comandos que se han utilizado en la práctica anterior, y casi todos los que hay, no son otra cosa que ficheros ejecutables (programas), aunque hay algunas excepciones como por ejemplo `cd`, `type` y `source`, que son funcionalidades ofrecidas por la *shell* (pero por ahora podemos obviar esta diferencia).

Cuando escribimos un comando y pulsamos ENTER, la *shell* busca que exista un fichero con este nombre en alguno de los directorios que conoce, y si lo encuentra lo ejecuta (lo cual implica crear un nuevo proceso donde se ejecuta ese programa mientras que el proceso de la shell se queda esperando a que termine). Si no encuentra ningún fichero con el nombre indicado nos devolverá un error indicando que no encuentra el programa correspondiente. Por ejemplo:

```
$ pwd
/home/alumno
$ c
c: command not found
```

Con el comando `which` se puede consultar la localización del fichero correspondiente a un comando. Por ejemplo:

```
$ which grep
/bin/grep
$ ls -l /bin/grep
-rwxr-xr-x 1 root root 211224 abr 29  2016 /bin/grep
```

Con esto se puede comprobar que el comando `grep` es un fichero que se encuentra en la carpeta `/bin` y tiene activado el permiso de ejecución para todos los usuarios. Busque en qué directorios se encuentran los comandos `ls`, `grep`, `which` y `java`. ¿Se encuentran todos en la misma carpeta? ¿Ve alguna diferencia notable entre los ficheros?

Adicionalmente, en vez de delegar a la *shell* la tarea de encontrar el programa que debe ejecutar, se puede indicar de forma explícita indicando la ruta absoluta o relativa hasta el fichero correspondiente. Ejecute los comandos y compruebe si detecta alguna diferencia:

```
$ cd /bin
$ echo "esto es una prueba" | /bin/grep "es"
$ echo "esto es una prueba" | ./grep "es"
```

Como se ha comentado antes, la shell busca estos comandos en una lista de directorios conocidos. Algunas rutas habituales son:

```
/usr/bin
/bin
```

```
/usr/local/bin  
/home/<usuario>/local/bin
```

No obstante, la lista exacta y el orden de prioridad de estas rutas depende de la configuración de la *shell*. La lista que su *shell* utiliza para buscar los programas se almacena en la variable de entorno² `PATH`. Pruebe a imprimirla para ver su contenido:

```
$ echo $PATH
```

De esta forma, la *shell* permite ejecutar como comandos aquellos ficheros que tengan habilitado el permiso de ejecución. No obstante, para que el fichero pueda ejecutarse correctamente debe ser un binario en código máquina o un programa que el sistema operativo pueda interpretar³.

La gran mayoría de los comandos que hemos utilizado hasta el momento son binarios compilados, habitualmente de programas escritos en C o C++. Por ejemplo, mire el contenido del comando `grep`:

```
$ head /bin/grep
```

Debería ver una serie de caracteres no imprimibles (habitualmente algo similar a ) junto con alguna que otra cadena de texto legible.

Por el contrario, también existen comandos cuyo código fuente se interpreta en cada ejecución. Es posible que se encuentre con programas escritos en Python, pero lo más habitual es que sean programas directamente interpretados por una *shell*. Mire por ejemplo el contenido del comando `which`.

```
$ head /bin/which
```

Aquí verá texto completamente legible. Además, si se fija en la primera línea del programa verá una secuencia de texto habitual llamada *shebang*⁴. Esta línea inicia con los caracteres `#!` y continúa con la ruta absoluta al programa que debe interpretar las líneas posteriores. En este caso, `/bin/sh`, nos indica que el programa se interpretará por la *shell* `sh`.

Dependiendo del lenguaje en el que se haya escrito el programa, encontrará distintos contenidos en esta línea inicial. ¿Qué esperaría encontrar en ficheros que empiecen con las siguientes líneas?

```
#!/bin/sh  
#!/bin/bash  
#!/usr/bin/python3
```

² <https://www.hostinger.com/es/tutoriales/variables-en-bash>

³ <https://www.freecodecamp.org/espanol/news/lenguajes-compilados-vs-interpretados/>

⁴ <https://es.wikipedia.org/wiki/Shebang>

3. Compilando y ejecutando un programa

En lo restante de esta práctica nos vamos a centrar en la programación de comandos en Java. Por desgracia, la versión de Java que hay instalada en los ordenadores del laboratorio no permite la ejecución de ficheros utilizando *shebang* (sí le funcionará en los ordenadores virtuales de la universidad con el shebang `#!/usr/share/java --source 11`). A su vez, Java no es un lenguaje que se compile a código máquina, por lo que su compilación tampoco genera binarios directamente ejecutables por el sistema operativo. En su lugar, el compilador de Java genera un fichero intermedio conocido como *bytecode*, que posteriormente es interpretado y ejecutado por la máquina virtual de java (JVM)⁵. Esto al final quiere decir que lo que tendremos que lanzar para ejecutar nuestros programas Java es el propio programa java, que es el que contiene la JVM, dándole el *bytecode* de nuestro programa.

Utilizaremos como primer ejemplo el siguiente programa. Abra un editor de texto de su preferencia, copie el código que encontrará a continuación y guarde el fichero con el nombre `Echo.java`. Dedique un momento a leer y entender el código del programa.

```
class Echo {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.print(args[i]);
            if (i < args.length - 1) System.out.print(" ");
        }
        System.out.println();
    }
}
```

Podemos compilar el programa utilizando el comando `javac`.

```
$ javac Echo.java
```

Si la compilación ha sido satisfactoria, deberíamos ver que se ha generado un fichero `Echo.class`. En este fichero se habrá almacenado el *bytecode* generado por el compilador. Al ejecutar el comando `java`, éste leerá el fichero `Echo.class` e interpretará su contenido en la JVM.

Pruebe los siguientes comandos:

```
$ java Echo
$ java Echo a b c
$ java Echo a "b c"
```

¿Cuál es la diferencia entre el segundo y tercer ejemplo?

⁵ <https://www.geeksforgeeks.org/java/compilation-execution-java-program/>

Para conseguir un comando ejecutable podemos definir un pequeño *script* en bash que actúe como envoltorio del programa:

```
#!/bin/bash
java_file=<abspath_to_java_file>

tmpdir=$(mktemp -d)
javac -d $tmpdir $java_file
java -cp $tmpdir $(basename $java_file) "$@"
rm -r $tmpdir
```

Este script realiza las siguientes acciones:

1. Define una variable `java_file` donde se deberá especificar la ruta absoluta al fichero que contiene las fuentes del programa de Java.
2. Crea un directorio temporal mediante `mktemp -d` en el que guardar el `.class` resultante de la compilación. Guarda la ruta absoluta de este directorio en la variable `tmpdir`.
3. Ejecuta `javac` para compilar el programa indicado y almacenar el `.class` en el directorio temporal.
4. Ejecuta el programa `java`, indicándole el directorio en el que buscar el fichero `.class`, el nombre del programa, y le traslada cualquier argumento con el que se ejecute el script de bash.
5. Borra el directorio temporal que se había creado.

Cree un fichero llamado `Echo` con este programa, asignando correctamente el valor de la variable `java_file` para apuntar al fichero `Echo.java`. Permita la ejecución de este fichero con `chmod` y muévelo a una de las rutas en las que su *shell* busca los comandos. Tenga en cuenta que no tendrá permisos de escritura en la mayoría de los directorios configurados en la shell. Pruebe los ejemplos anteriores pero esta vez sin necesidad de invocar `java` explícitamente.

```
$ Echo
$ Echo a b c
$ Echo a "b c"
```

Nota: Definir pequeños scripts de bash como este pueden ayudarle de forma simple a agilizar tareas que realiza frecuentemente. También son una buena forma de traer las prácticas preparadas y que sirvan como apuntes si los necesita posteriormente.

4. Implementando grep

En esta sección se pide implementar un programa en Java que replique el funcionamiento básico del programa grep. En concreto, trate de replicar los siguientes ejemplos:

```
$ cat /etc/services | grep web
$ find /etc | grep java
$ strings /bin/* | grep 'Free Software Foundation'
```

En estos ejemplos el programa leerá texto de la entrada estándar e imprimirá por pantalla aquellas líneas que contengan el texto indicado como argumento.

En Java, para leer texto de la entrada estándar puede utilizar la clase Scanner⁶. Se proporciona el siguiente fragmento de código a modo de ayuda:

```
import java.util.Scanner;
Scanner scanner = new Scanner(System.in);
while (sc.hasNextLine()) {
    String line = scanner.nextLine();
}
```

Deberá poder ejecutar su programa de la siguiente manera:

```
$ cat /etc/services | Grep web
$ find /etc | Grep java
$ strings /bin/* | Grep 'Free Software Foundation'
```

Punto de control 1 (20%): Muestre al profesor el programa indicado.

Nota: Para este punto de control no necesita que su programa resalte con color el texto buscado, pero si le interesa puede tratar de añadir esta funcionalidad, no es complicado de hacer. Únicamente necesitará introducir un código de color⁷ antes del texto y un código de reset después. Estos códigos serán interpretados por la shell al imprimir el texto. Desde Java tendrá que sustituir el carácter `\e` por `\u001B`.

⁶ <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>

⁷ <https://gist.github.com/JBlond/2fea43a3049b38287e5e9cefc87b2124>

5. Simplificando *ifconfig*

Es habitual que para realizar una tarea se dispongan de los comandos para obtener la información necesaria, pero en ocasiones en un formato que no es el más cómodo para lo que se necesita (posiblemente la salida incluye más información que en ese momento no le resulta relevante), o en un formato que no es posible transmitir directamente a otra herramienta que necesita utilizar.

En prácticas anteriores ha utilizado `ifconfig` para listar las interfaces de red de su equipo. También habrá necesitado ver la dirección MAC de cada interfaz.

Implemente un programa que lea por *stdin* la salida de `ifconfig` y genere a su salida una lista de interfaces y direcciones MAC. Puesto que el programa de Java no esperará argumentos, puede pasarle todos los argumentos recibidos directamente a `ifconfig`. Ignore las interfaces que no tienen dirección MAC.

Deberá poder ejecutar su programa de la siguiente forma y obtener una salida similar a la del ejemplo:

```
$ simple-ifconfig
eth0 00:0d:88:cd:6e:c8
eth1 00:0d:88:cd:6e:c9
eth2 00:0d:88:cd:6e:ca
```

```
$ simple-ifconfig -a
eth0 00:0d:88:cd:6e:c8
eth1 00:0d:88:cd:6e:c9
eth2 00:0d:88:cd:6e:ca
eth3 00:0d:88:cd:6e:cb
```

Punto de control 2 (20%): Muestre al profesor el programa indicado.

6. Procesando tráfico

En prácticas anteriores ha usado `tshark` para capturar y ver tráfico desde la terminal. En ocasiones resulta necesario procesar y analizar este tráfico para obtener un resumen que permita extraer conclusiones sobre los equipos que se están comunicando en la red.

Conecte varias de las interfaces de su equipo a un conmutador. Genere tráfico con distintos valores de *ethertype* y variedad en las direcciones origen y destino. Inicie *tshark* con las opciones mostradas a continuación para generar un fichero de texto con la captura realizada.

```
$ sudo tshark -i eth0 -nl -T fields -e frame.time_relative -e eth.src -e eth.dst -e eth.type
6.469932427 08:00:27:61:8f:36 ff:ff:ff:ff:ff:ff 0x0800
6.550590993 00:26:86:00:00:00 ff:ff:ff:ff:ff:ff 0x0806
6.855973089 08:00:27:61:8f:36 ff:ff:ff:ff:ff:ff 0x88b5
7.144841975 28:f5:2b:b4:21:66 08:00:27:61:8f:36 0x88b5
7.761027524 aa:aa:aa:aa:aa:aa cc:cc:cc:cc:cc:cc 0x88b5
7.798099335 28:f5:2b:b4:21:66 ff:ff:ff:ff:ff:ff 0x0800
```

A continuación se le pedirá implementar distintos tipos de análisis a partir de este fichero de texto. El programa siempre recibirá como primer argumento la ruta al fichero de texto y un segundo argumento que indicará qué análisis debe realizar. Cada tipo de análisis puede permitir argumentos adicionales que serán interpretados de forma diferente. Si se especifica un tipo de análisis no soportado, deberá emitir un mensaje de error indicándolo.

Nota: *El programa deberá ir incrementando las funcionalidades que ofrece a lo largo de los distintos puntos de control, pero siempre manteniendo soporte para las opciones anteriores. **No elimine el código de un punto de control al pasar al siguiente.***

6.1. Contador de tramas por origen

Permita que su programa responda al análisis count-src-mac. Deberá leer el contenido del fichero de texto e imprimir una lista con las direcciones MAC origen junto al número de paquetes enviados desde cada una. Por ejemplo:

```
$ java TSharkAnalyzer.java capture.txt count-src-mac
00:26:86:00:00:00 3
aa:aa:aa:aa:aa:aa 8
08:00:27:61:8f:36 36
28:f5:2b:b4:21:66 14
```

Añada un segundo argumento opcional a su programa. Si se indica, este será interpretado como un *ether type* por el cual se filtrará el tráfico y se ignorarán el resto de tramas en el resumen mostrado por el programa.

```
$ java TSharkAnalyzer.java capture.txt count-src-mac 0x88b5
aa:aa:aa:aa:aa:aa 5
08:00:27:61:8f:36 30
28:f5:2b:b4:21:66 11
```

Punto de control 3 (20%): Muestre al profesor el programa indicado.

6.2. Contador de tramas por conversación

Permita que su programa responda al análisis count-conversations. Deberá leer el contenido del fichero de texto e imprimir una lista con las parejas de direcciones MAC origen y destino, junto al número de paquetes enviados de un extremo al otro. Por ejemplo:

```
$ java TSharkAnalyzer.java capture.txt count-conversations
00:26:86:00:00:00 -> ff:ff:ff:ff:ff:ff 3
aa:aa:aa:aa:aa:aa -> cc:cc:cc:cc:cc:cc 8
08:00:27:61:8f:36 -> ff:ff:ff:ff:ff:ff 15
28:f5:2b:b4:21:66 -> ff:ff:ff:ff:ff:ff 21
08:00:27:61:8f:36 -> 28:f5:2b:b4:21:66 13
28:f5:2b:b4:21:66 -> 08:00:27:61:8f:36 4
```

Implemente ahora una opción adicional `-m` o `--merge-directions` que permita combinar los contadores de una misma conversación independientemente de la dirección de las tramas. Así, no será relevante cuál de las dos direcciones era origen y cual es destino. Es decir, un paquete enviado desde A -> B, se deberá contar junto los paquetes enviados de B -> A. Por ejemplo:

```
$ java TSharkAnalyzer.java capture.txt count-conversations
00:26:86:00:00:00 <-> ff:ff:ff:ff:ff:ff 3
aa:aa:aa:aa:aa:aa <-> cc:cc:cc:cc:cc:cc 8
08:00:27:61:8f:36 <-> ff:ff:ff:ff:ff:ff 15
28:f5:2b:b4:21:66 <-> ff:ff:ff:ff:ff:ff 21
08:00:27:61:8f:36 <-> 28:f5:2b:b4:21:66 17
```

Nota: Puede serle de utilidad el método `String.compareTo`⁸.

Punto de control 4 (25%): Muestre al profesor el programa indicado.

⁸ <https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareTo-java.lang.String->

6.3. Destinos más frecuentes

Permita que su programa responda al análisis top-dst. Deberá leer el contenido del fichero de texto e imprimir la dirección MAC del destino que más paquetes ha recibido, junto al número de paquetes.

```
$ java TSharkAnalyzer.java capture.txt top-dst  
ff:ff:ff:ff:ff:ff 73
```

Implemente una opción adicional -n <num> que permita indicar el número de destinos máximos a mostrar, priorizando siempre los que más tramas han recibido. Por ejemplo:

```
$ java TSharkAnalyzer.java capture.txt top-dst -n 3  
ff:ff:ff:ff:ff:ff 73  
28:f5:2b:b4:21:66 13  
cc:cc:cc:cc:cc:cc 8
```

Implemente otra opción -e <mac> o --exclude-dst <mac> que permita indicar una dirección MAC que deba ser excluida del procesado. Esta opción deberá poder indicarse múltiples veces. Por ejemplo:

```
$ java TSharkAnalyzer.java capture.txt top-dst -n 3 -e ff:ff:ff:ff:ff:ff -  
-exclude-dst cc:cc:cc:cc:cc:cc  
28:f5:2b:b4:21:66 13  
08:00:27:61:8f:36 4
```

Punto de control 5 (15%): Muestre al profesor el programa indicado.
