Comandos Básicos (uso básico de la terminal)

Apuntes de la clase

Enrique Calderón, Francisco Galindo

Curso de SysAdmin, Semestre 2025-1

Índice

1 Ideas preliminares

Este tema es una guía/repaso para varios de los comandos que se utilizarán en el resto del curso. Es muy posible que ya estén familiarizados con los conceptos mostrados a continuación, pero el objetivo es asegurarse de que todos entiendan qué está ocurriendo cuando se empiecen a escribir *scripts* o se hagan configuraciones al sistema mediante la línea de comandos.

1.1 La *shell*

Cuando se abre una terminal y aparece el *prompt* (`$`), puedes escribir comandos y estos se ejecutarán al presionar Enter:

```
$ whoami
juanito
```

El programa encargado de interpretar qué programas debe ejecutar a partir de lo que escribas en la terminal se le llama *shell*. La *shell* es sólo un programa al igual que la gran mayoría de los comandos que ejecuta. El objetivo de la *shell* es actual como un intermediario entre el *kernel* del sistema operativo y la persona que utiliza la computadora.

La *shell* por defecto en Debian es `bash`, aunque existen muchas otras, como `zsh`, `csh`, `tcsh`, `dash`, etc.

1.2 Sintaxis de la ejecución de un comando en la *shell*

Un comando generalmente empieza indicando el nombre de un programa que será ejecutado, seguido de distintos *argumentos* que le dicen a ese programa cómo debe comportarse y sobre qué operar. Los distintos argumentos se indican separados por espacios. Los argumentos que modifican el comportamiento por defecto de un programa suelen empezar con un guión (`-`), a estos se les suele conocer como banderas.

Un ejemplo de un comando es el siguiente:

```
$ cat /etc/passwd
```

Este comando hace que se ejecute el programa `cat`, y manda como argumento `/etc/passwd`. `cat` está programado para abrir el archivo con la ruta indicada en el argumento y mostrar sus contenidos en pantalla. Así pues, lo que hace el comando es mostrar los contenidos del archivo `/etc/passwd`.

1.3 Entrada y salida estándar

Un proceso[1] puede utilizar *streams* de entrada y salida (*I/O*) para leer y escribir información (se usa un *stream* de entrada para leer datos y uno de salida para escribir datos). Estos *streams* pueden ser archivos, dispositivos, una ventana de terminal, entre otros (por ejemplo, como se verá al revisar las *pipes*, un *stream* de salida de un proceso puede utilizarse como *stream* de entrada de otro). Cuando los *streams* de entrada o salida son una ventana de terminal, se está hablando de la entrada y salida estándar.

En otras palabras, cuando escribes datos a un programa a través de la terminal de un programa, estás escribiendo en su *stream* de **entrada estándar**. Cuando el programa escribe algo en la terminal, está utilizando su *stream* de **salida estándar**.

2 Catálogo de comandos útiles

2.1 Algunos comandos elementales

- `whoami`: Muestra el nombre de usuario de quien que ha ejecutado el comando:

  ```
  $ whoami
  juanito
  ```

- `man`: Muestra el manual de algún programa o biblioteca:

  ```
  $ man man
  ```

[1] Puedes pensar en un proceso como un programa en ejecución

Puedes navegar el manual con las flechas direccionales o con los comandos de movimiento de `vi` (pronto se hablará sobre `vi`). Se sale con `q`.

Si no estás seguro de cómo se utiliza un comando, trata de ver si tiene una página de manual y léela, te puede ahorrar una búsqueda en internet.

- `pwd`: Muestra el directorio actual de trabajo, es decir, el directorio en el que se está ejecutando el programa. Si se ejecuta desde la terminal, muestra el directorio en el que se encuentra el usuario que ejecutó el comando:

  ```
  $ pwd
  /home/juanito
  ```

- `ls`: Su función principal es mostrar el contenido de un directorio. Por defecto trabaja en el directorio actual de trabajo. El comando puede recibir muchas opciones, como `-l` para mostrar el contenido en formato largo, o `-a` para mostrar archivos ocultos[2].

- `cat`: Imprime en la salida estándar los contenidos de cada uno de los archivos indicados en sus argumentos, con**cat**enados uno después del otro:

  ```
  $ cat arch1 arch2 arch3
  ```

- `touch`: Suele utilizarse para crear un nuevo archivo vacío:

  ```
  $ touch archivo
  ```

  En realidad, este programa se encarga de actualizar la fecha (*timestamp*) de modificación de un archivo. Este comando tiene el efecto secundario de crear el archivo en cuestión en caso de que no existiera anteriormente.

- `mkdir`: Crea un nuevo directorio en la ruta indicada:

  ```
  $ mkdir nuevodir
  ```

- `cd`: Cambia el directorio actual de trabajo hacia la ruta indicada

  ```
  $ pwd
  /home/juanito
  $ ls
  dir
  $ cd dir
  $ pwd
  /home/juanito/dir
  ```

  Si se omite una ruta de destino, el comando `cd` te manda a tu directorio `home`.

- `cp`: Este comando se encarga de copiar archivos. Por ejemplo, ejecutar lo siguiente:

  ```
  $ cp original copia
  ```

  Copiaría los contenidos del archivo `original` y los escribiría en un nuevo archivo llamado `copia`. En general, el primer argumento indica la ruta hacia un archivo de origen, mientras que el segundo indica la ruta hacia el archivo de destino.

[2]Los archivos ocultos tienen un nombre que empieza con un punto (`.`). Estos archivos no se muestran por defecto para evitar congestionar la pantalla con archivos irrelevantes al momento de navegar el sistema de archivos.

Con la bandera `-r`, se puede hacer una copia recursiva, lo que significa que, si se utiliza con un directorio como origen, también se copiarán todos sus contenidos.

- `mv`: Es similar al comando `cp`, pero este tiene el objetivo de mover un archivo:

  ```
  $ mv /ruta/a/archivo /nueva/ruta/a/archivo
  ```

  También suele utilizarse para renombrar archivos:

  ```
  $ mv antes despues
  ```

- `rm`: Elimina un archivo. Esta eliminación es definitiva (en el sentido de que no hay una "papelera de reciclaje").

  ```
  $ rm archivo
  ```

  Con la bandera `-r`, puede eliminarse un directorio así como todos sus contenidos de manera recursiva:

  ```
  $ rm -r directorio
  ```

- `echo`: Este comando imprime todos sus argumentos a la salida estándar:

  ```
  $ echo Hola, mundo
  Hola, mundo
  ```

- `find`: Permite realizar búsquedas detalladas de archivos. Por ejemplo, este comando:

  ```
  $ find dir -name '*apunte*.pdf"
  ```

  Busca dentro del directorio `dir` aquel archivo cuyo nombre contenga en algún lugar la palabra `apunte` (nótense la *wildcards*), y cuya extensión sea `.pdf`.

- `wc`: Cuenta las palabras, caracteres y líneas de un archivo. Con la bandera `-l`, cuenta sólo las líneas.

- `diff`: Muestra las diferencias entre dos archivos de texto:

  ```
  $ diff arch1 arch2
  ```

- `less`: Es un paginador, lo puedes utilizar para leer un archivo de texto largo de manera ligeramente más cómoda. Es el programa con el que se visualizan las páginas de manual.

- `head` y `tail`: Estos comandos sirven para mostrar porciones de un archivo. `head` sirve para mostrar el inicio de un archivo. Por defecto, muestra las primeras dos líneas, aunque con la opción `-n` puede indicarse el número de líneas que se mostrarán:

  ```
  $ head -n 5 arch1 # Muestra las primeras 5 líneas de arch1
  ```

  El comando `tail` sirve para mostrar el final de un archivo. Las opciones funcionan de manera análoga a `head`. Adicionalmente, si se utiliza la opción `+n`, se indica la línea del archivo a partir de la cuál se va a imprimir el archivo.

- `sort`: Ordena un archivo, muestra el resultado ordenado en la salida estándar

  ```
  $ sort archivo
  ```

- `tar`: Una utilidad para compresión y descompresión de datos. Suele utilizarse para descomprimir los archivos comprimidos que distribuyen los desarrolladores de software para instalar sus programas.
- `curl`: Hace peticiones hacia alguna URL. Puede utilizarse para descargar archivos desd eun servidor remoto. Para hacer esto último, se necesitan las banderas `-LO`
- `du`: Muestra el uso en disco de un archivo.

2.2 *Wildcards*

La *shell* puede buscar patrones simples en nombre de archivos para evitar tener que escribir de más. El más simple es el asterisco (`*`), que le dice a *shell* que busque cualquier cantidad de caracteres arbitrarios. Por ejemplo:

```
$ ls
arch1 arch2 arch3 texto
$ echo *
arch1 arch2 arch3 texto
$ echo arch*
arch1 arch2 arch3
```

- `palabra*` se expande a todos los nombres de archivos que terminen con 'palabra'.
- `*palabra` se expande a todos los nombres de archivos que empiecen con 'palabra'.
- `*palabra*` se expande a todos los nombres de archivos que contengan 'palabra'.

2.3 Comandos para obtener información del Sistema

- `uname`: muestra información como el `hostname`, version del *kernel* y la arquitectura del procesador.
- `uptime`: Muestra el tiempo que la computadora ha estado funcionando.
- `free`: Muestra el uso de memoria RAM y de *swap*.
- `df`: Muestra el uso de los discos de la computadora.

2.4 Comandos para procesamiento de texto

- `grep`: Permite la búsqueda de cierta cadena en un archivo. Te permite encontrar las partes del archivo que contienen (o que no contienen) la cadena de interés. Extremadamente útil para extraer información útil de un archivo grande.

  Imprime las líneas de un archivo que coinciden con una expresión. Por ejemplo, si se quiere encontrar al usuario `root` en el archivo `/etc/passwd`:

  ```
  $ grep root /etc/passwd
  ```

  Puede utilizarse en varios archivos:

  ```
  $ grep root /etc/*
  ```

Puede utilizarse la opción `-i` para hacer búsquedas que no sean sensibles a mayúsculas o minúsculas, `-v` para invertir la búsqueda (sólo muestra las líneas que no coinciden con la búsqueda).

`grep` entiende expresiones regulares con la opción `-E` (también conocido como el comando `egrep`).

Un poco de sintaxis de expresiones regulares:

- `c` coincide con el símbolo c.
- `\c` coincide con el símbolo literal c.
- `.` cualquier símbolo.
- `^` final de la línea
- `$` inicio de la línea
- `[abc...]` lista, coincide con cualquiera de a, b, c...
- `[^abc...]` lista negada, coincide con cualquier símbolo excepto a, b, c...
- `[0-9a-zA-Z]` range of characters 0-9 and a-z,A-Z
- `r1|r2` alternación: coincide con r1 o r2.
- `r1r2` concatenación: coincide con r1 seguido de r2.
- `r+` coincide con una o más r's.
- `r*` coincide con cero o más r's.
- `r{2,5}`: coincide con entre 2 y 5 r's consecutivas.
- `r?` : coincide con zero or one r's.

- `sed`: Permite filtrar y editar flujos de texto. Generalmente se usa para hacer modificaciones relativamente sencillas en archivos o en la salida de algún otro programa. Se utilizan expresiones regulares para definir qué partes del texto modificar o mostrar.
- `awk`: ¡Es lenguaje de programación! A diferencia de Python, C y el resto, `awk` se centra en el procesamiento de texto.

2.4.1 Ejemplos

```
awk '{print $2}' arch.txt
```

Imprime la segunda columna cada línea del archivo `arch.txt`.

```
awk  awk  -F ":" '{print $2}' arch.txt
```

Igual, pero el separador de columnas es el caracter `:`.

```
awk '/una regex/{print $2}' arch.txt
```

Imprime la segunda columna de las líneas que coincidan con la *regex* `/una regex/`

```
awk '$1 ~ /some regexp/{print $2}' arch.txt
```

Imprime la segunda columna de las líneas que coincidan con la *regex* `/una regex/`

3 Redirección de entrada y salida

La mayoría de los comandos que se han utilizado hasta ahora, mientras se están ejecutando, muestran su salida en la terminal, algunos también reciben información de este modo. En

sistemas basados en *UNIX*, es posible reemplazar esta terminal con un archivo tanto para la entrada como para la salida de datos en un programa[3][4].

3.1 Redirección de salida

Por ejemplo, si listamos los contenidos de la carpeta `home` de un usuario, es posible que veamos algo como esto:

```
$ ls -l
.  ..  .bash_logout  .bashrc  .config  .lesshst  .profile  .ssh
```

Esta salida es efímera, lo que significa que cuando cerremos la terminal o cuando limpiemos la pantalla, se perderán los resultados del programa. Si lo que se quiere es almacenar estos resultados a largo plazo, o para analizarlos después, puede hacerse que estos archivos se guarden en un archivo en lugar de mostrarse en la terminal:

```
$ ls -l >mi-home.txt
```

Y si vemos los contenidos del nuevo archivo `mi-home.txt`:

```
$ cat mi-home.txt
.  ..  .bash_logout  .bashrc  .config  .lesshst  .profile  .ssh
```

Aquí, el símbolo `>` es el que indica que la salida del programa a la izquierda del mismo será redirigida al archivo indicado a la derecha del símbolo. Nótese que la salida del programa `ls` no se mostró en la terminal para nada.

El archivo al cual se hace referencia al redirigir la salida es creado en el caso de que no existiera anteriormente. Si el archivo ya existía, **los contenidos son sobreescritos por completo**. Así que cualquier cosa que estuviera almacenada en el archivo anteriormente se pierde.

Si no se desea sobreescribir los datos almacenados en el archivo de destino, puede utilizarse el símbolo `>>`, que opera de manera muy similar a `>`, con la diferencia de que `>>` agrega los nuevos datos al final de los ya existentes. Un ejemplo del uso de `>>` es el siguiente: en un principio concatenas varios archivos utilizando `cat`.

```
$ cat apunte1 apunte2 apunte3 >apuntes # Utilizar ">>" funciona igual
```

Pero posteriormente te das cuenta de que no tomaste en cuenta al archivo `apunte4`, en lugar de ejecutar el comando entero otra vez, puedes simplemente agregar el faltante con `>>`:

```
$ cat apunte4 >>apuntes
```

3.2 Redirección de entrada

Si el símbolo `>` nos permite redirigir la salida de un programa, resulta natural que con `<` se pueda redirigir la entrada. Por ejemplo, si se necesita ordenar la lista de alumnos de un curso, puede utilizarse el siguiente comando:

[3] Los ejemplos mostrados en estas últimas dos secciones están profundamente inspiradas en los mostrados en el libro *The UNIX programming environment* de Kernighan y Pike.

[4] Con entrada y salida "desde la terminal", se está haciendo referencia a la entrada, salida y error estándar (`stdin` y `stdout`, `stderr`).

```
$ sort <alumnos.txt
Alberto
...
Yael
Zaír
```

El resultado es el mismo que el que hubiéramos obtenido con el comando `sort alumnos.txt`, sin embargo, la diferencia aquí es que, cuando no se utiliza redirección, el programa `sort` se encarga de abrir y leer el archivo para posteriormente ordenarlo. Cuando uno redirige con `<`, es la *shell* la que se encarga de abrir y leer el archivo, así que desde la perspectiva de `sort`, hay una persona escribiendo los nombres uno a uno.

Esta diferencia tiene la poderosa implicación de que ahora podemos utilizar archivos como datos de entrada para cualquier programa que lea desde la terminal, independientemente de que haya sido programado para ello.

4 Tuberías (`pipes`)

El concepto de redirección invita la posibilidad de ejecutar un comando, guardar su salida en un archivo, y luego utilizar ese archivo para alimentar a otro comando. Este proceso se vuelve tedioso muy rápido, por lo que se creó el concepto de *pipes*. Una *pipe* es una manera de conectar la salida estándar de un programa con la entrada estándar de otro. Se le llama *pipeline* a la conexión de dos o más programas por medio de *pipes*. Algunos ejemplos de pipes son:

```
$ ls | wc -l # Cuenta los archivos
```

```
$ du -sh ./* | sort -rh | head -5 # Muestra top 5 archivos pesadosraquí
```

Los programas dentro de una *pipeline* se ejecutan al mismo tiempo, de manera concurrente, no secuencialmente como parecería a primera vista. Ejecuta `cat | grep` para verificarlo.

Lo increíblemente poderoso de las tuberías es que permite la colaboración entre muchos programas de manera completamente transparente, sin necesidad de que estos estén programados explícitamente para interactuar entre sí. Los programas ni siquiera necesitan saber que están dentro de una *pipeline*. A partir de *pipelines* pueden generarse comportamientos extremadamente complejos con una serie de programas simples.