# Comandos Básicos (uso básico de la terminal)

Enrique Calderón, Francisco Galindo
Estudiante de Ingeniería en Computación

*Universidad Nacional Autónoma de México*
*Facultad de Ingeniería*

Curso de SysAdmin, Julio de 2024

# Recordatorio

También hay apuntes correspondientes a esta presentación. Ahí, algunos conceptos presentados (sobre todo redirección y *pipes*) estarán explicados en mayor detalle que en las diapositiva con el objetivo de no sobrecargar la presentación.

# 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.

# La *shell*

El programa encargado de interpretar qué programas debe ejecutar a partir de lo que escribas en la terminal se le llama *shell*.

Es sólo un programa como cualquier otro. Su objetivo es actuar 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.

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

Un comando generalmente empieza indicando el nombre de un programa seguido de distintos *argumentos* que indican comportamiento. 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
```

# 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**.

# Algunos comandos elementales

- `whoami`: Muestra el nombre de usuario de quien que ha ejecutado el comando:
- `man`: Muestra el manual de algún programa o biblioteca:

  ```
  $ man man
  ```

  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`.
- `pwd`: Muestra el directorio actual de trabajo.

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

# Algunos comandos elementales

- `ls`: Su función principal es mostrar el contenido de un directorio.
- `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
  ```

# Algunos comandos elementales

- `mkdir`: Crea un nuevo directorio en la ruta indicada:
- `cd`: Cambia el directorio actual de trabajo hacia la ruta indicada

  Si se omite una ruta de destino, el comando `cd` te manda a tu directorio `home`.
- `cp`: Este comando se encarga de copiar archivos.

  ```
  $ cp original copia
  ```

  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.

# Algunos comandos elementales

- `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:

# Algunos comandos elementales

- `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`.

# Algunos comandos elementales

- `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.

# Algunos comandos elementales

- `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 diez líneas.

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

# Algunos comandos elementales

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

  ```
  $ sort archivo
  ```

- `tar`: Una utilidad para compresión y descompresión de datos.
- `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.

# *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:

- `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'.

# 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.

# 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 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

# 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.

# Comandos para procesamiento de texto

- `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.

# Comandos para procesamiento de texto

- `awk`: ¡Es lenguaje de programación! A diferencia de Python, C y el resto, `awk` se centra en el procesamiento de texto.

## Ejemplos

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

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

```
awk  awk  -F ":" '{print $2}' arch.txt
awk '/una regex/{print $2}' arch.txt
awk '$1 ~ /some regexp/{print $2}' arch.txt
```

# Redirección de entrada y salida

## Redirección de salida

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
```

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

## Redirección de salida

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

# 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
```

# Tuberías (`pipes`)

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.