El intérprete de comandos o “Shell”

¿Qué es el Shell?

Las computadoras modernas tienen una variedad de interfaces mediante las cuales operamos con ellas; sofisticadas interfaces gráficas de usuario, interfaces de voz e incluso AR/VR (Realidad Aumentada / Realidad Virtual) ya se ven en todas partes. Éstas son excelentes para el 80% de los casos, pero a menudo están fundamentalmente restringidas en lo que nos permiten hacer: no es posible presionar un botón que no existe o dar un comando de voz que no ha sido programado. Para aprovechar al máximo las herramientas que proporciona la computadora, tenemos que ir “a la vieja escuela” y caer en una interfaz de texto: el intérprete de comandos, más conocido como “shell”.

Casi todas las plataformas sobre las que podemos trabajar tienen algún tipo de intérprete de comandos; muchas de ellas tienen incluso varios intérpretes para elegir. Si bien pueden variar en los detalles, en esencia son más o menos lo mismo: nos permiten ejecutar programas, proveerles entradas e inspeccionar las salidas de una forma semi-estructurada.

En esta clase, nos enfocaremos en el Bourne Again SHell, o “bash” para abreviar. Este es uno de los shells más utilizados, y su sintaxis es similar a la que verás en muchos otros intérpretes de comandos. Para iniciar un shell *prompt* (el lugar donde escribir los comandos), primero hace falta una *terminal*. Tu computadora probablemente ya tenga un programa terminal instalado, pero igual podés instalar uno nuevo con bastante facilidad.

Usando el shell

Al iniciar la terminal, verás un texto o *prompt* que suele parecerse a esto:

```
missing:~$
```

Esta es la interfaz textual principal del shell. Nos dice que nos encontramos en la máquina llamada `missing` y que nuestro “directorio de trabajo actual”, o dónde se encuentra actualmente, es `~` (abreviatura de “home”). El `$` indica que no eres el usuario root (veremos esto más adelante). Luego de este indicador, es posible escribir un *comando*, que luego será interpretado por el shell. El comando más básico que podemos solicitar al intérprete es ejecutar un programa:

```
missing:~$ date
vie 10 ene 11:49:31 -03 2020
missing:~$
```

Aquí hemos ejecutado el programa `date`, que (como era de esperar) muestra la fecha y hora actuales. El shell luego nos pide otro comando para ejecutar. También podemos ejecutar un comando con *argumentos*:

```
missing:~$ echo Hola
Hola
```

En este caso, le pedimos al intérprete que ejecute el programa `echo` con el argumento `Hola`. El programa `echo` simplemente imprime los argumentos que recibe. El shell analiza el comando separándolo donde hay espacios en blanco y luego ejecuta el programa que figura en la primera palabra, pasando cada palabra posterior como un argumento que el programa puede utilizar. Si deseamos proporcionar un argumento que contenga espacios u otros caracteres especiales (por ejemplo, un directorio llamado “Mis fotos”), podemos rodear el argumento con comillas `'` o `"` (`"Mis Fotos"`), o bien escapar sólo los caracteres relevantes con el símbolo `\` (como en `Mis\ Fotos`).

Pero, ¿cómo sabe el shell dónde están los programas `date` o `echo`? Bueno, el shell es un entorno de programación, al igual que Pascal, Python, o Ruby, por lo que tiene variables, condicionales, bucles y funciones (¡próxima clase!). Cuando ejecutamos comandos en el intérprete, lo que realmente estamos haciendo es escribir un pequeño código fuente que el shell interpreta. Cuando pedimos al shell que ejecute un comando que no coincide con una de sus palabras clave de programación, el shell consulta una *variable de entorno* llamada `$PATH` que lista en qué directorios o carpetas el shell debe buscar los programas cuando recibe un nuevo comando:

```
missing:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
missing:~$ which echo
/bin/echo
missing:~$ /bin/echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Cuando ejecutamos el comando `echo`, el shell entiende que debe ejecutar el programa `echo`, y luego busca en toda la lista de directorios que figuran en `$PATH` (separados por `:`) un archivo con el nombre `echo`. Cuando lo encuentra, lo ejecuta (suponiendo que el archivo es *ejecutable*; más adelante volveremos a esto). Podemos averiguar qué archivo se ejecuta para cada nombre de programa usando el programa `which`. También podemos omitir `$PATH` por completo dando la ruta completa o *path* al archivo que queremos ejecutar.

Moviéndonos por el shell

Una ruta en el shell es una cadena de texto que menciona directorios o carpetas separados por el símbolo `/` en Linux y macOS o el símbolo `\` en Windows. En Linux y macOS, la ruta `/` es la "raíz" del sistema de archivos, dentro de la cual se encuentran todas las demás carpetas y archivos, mientras que en Windows hay una raíz por cada partición de disco (por ejemplo, `C:\`). En nuestra clase generalmente asumiremos que está utilizando un sistema de archivos Linux. Una ruta que comienza con `/` se llama *ruta absoluta*. Cualquier otra ruta es una *ruta relativa*. Las rutas relativas dependen de dónde uno está ubicado cuando ejecuta un comando (el directorio de trabajo actual), que podemos ver con el comando `pwd` y cambiar con el comando `cd`. En una ruta, el caracter `.` por sí solo hace referencia al directorio actual y la secuencia `..` hace referencia al directorio padre:

```
missing:~$ pwd
/home/missing
missing:~$ cd /home
missing:/home$ pwd
/home
missing:/home$ cd ..
missing:/$ pwd
/
missing:/$ cd ./home
missing:/home$ pwd
/home
missing:/home$ cd missing
missing:~$ pwd
/home/missing
missing:~$ ../../bin/echo hola
```

```
hola
```

En todo momento, nuestro *prompt* (el indicador en el intérprete de comandos) nos mantuvo informados sobre cuál era nuestro directorio de trabajo actual. Es posible configurar el prompt para que muestre todo tipo de información útil. Esto lo veremos en una clase posterior.

En general, cuando ejecutamos un programa, va a operar en el directorio actual salvo que le indiquemos lo contrario. Por ejemplo, generalmente buscará archivos allí y creará nuevos archivos allí si fuera necesario.

Para ver qué archivos y carpetas hay en un directorio dado, utilizamos el comando `ls` :

```
missing:~$ ls
missing:~$ cd ..
missing:/home$ ls
missing
missing:/home$ cd ..
missing:/$ ls
bin
boot
dev
etc
home
...
```

A menos que se proporcione un directorio como primer argumento, `ls` mostrará el contenido del directorio actual. La mayoría de los comandos aceptan argumentos y opciones (flags con valores) que comienzan con `-` para modificar su comportamiento. Por lo general, ejecutar un programa con la opción `-h` o `--help` (`/?` en Windows) mostrará un texto de ayuda que indica qué otras opciones están disponibles. Por ejemplo, `ls --help` nos dice:

```
  -l                         utiliza un formato de listado largo
```

```
missing:~$ ls -l /home
drwxr-xr-x 1 gabriel  users  4096 Jun 15  2019 missing
```

Esto nos da mucha más información sobre cada archivo o directorio existente. Primero, la `d` al principio de la línea nos dice que `missing` es un directorio. Luego siguen tres grupos de tres caracteres (`rwx`). Estos indican qué permisos tiene (es decir, qué acciones puede realizar) el dueño del archivo (`gabriel`) sobre el archivo `missing` , qué permisos tiene el grupo al que pertenece el archivo (`users`), y qué permisos tiene el resto de los usuarios del sistema sobre ese elemento, respectivamente. Un `-` en algún lado indica que ese sujeto (dueño, grupo o “todos los demás”) no tiene cierto permiso otorgado. En el ejemplo de arriba, solo el dueño del archivo (`gabriel`) puede modificar (`w`) el directorio `missing` (es decir, sólo `gabriel` puede agregar y/o eliminar archivos en ese directorio). Para poder acceder a un directorio, aquél usuario que desea accederlo debe poseer permisos de “búsqueda” (representado por la letra `x`) en ese directorio (y sus directorios padres). Para listar su contenido, un usuario debe tener permisos de lectura (`r`) en ese directorio. Así como sucede con los directorios, también es posible definir permisos sobre cualquier archivo: `r` para poder leerlo (Read), `w` para poder escribir sobre él (Write), y `x` para poder ejecutarlo (eXecute, en el caso de que ese archivo fuera un programa). Por ejemplo, todos los archivos en la carpeta `/bin` tienen el permiso `x` establecido para el último grupo (“los demás usuarios”), para que cualquier usuario pueda ejecutar esos programas.

Algunos otros programas útiles son `mv` (que se utiliza para cambiar el nombre o mover un archivo), `cp` (que se utiliza para copiar un archivo) y `mkdir` (que se utiliza para crear un nuevo directorio).

En todo momento que quieras *conocer más* sobre algún comando, sobre las opciones que soporta, entradas, salidas o sobre el funcionamiento general de cualquier programa, probá con el programa `man` . `man` toma como argumento el nombre de un programa y te muestra su *manual*. Para salir del manual pulsá la tecla `q` .

```
missing:~$ man ls
```

Conectando programas

En el shell, los programas tienen dos "flujos" primarios o `streams` asociados con ellos: su flujo de entrada (`standard input`) y su flujo de salida (`standard output`). Cuando un programa intenta solicitar una entrada, la solicita del flujo de entrada, y cuando muestra o imprime algo, lo hace en su flujo de salida. Comúnmente, la terminal actúa tanto como entrada como salida de un programa. Es decir, tu teclado actúa como entrada y tu pantalla como salida. Sin embargo, podemos redirigir esos flujos para que un programa reciba datos de otro o envíe datos a otro. A esto lo denominamos *redirección*.

La formas más simples de redirección son `< archivo.txt` y `> archivo.txt` . La primera redirige el flujo de entrada para que un programa tome datos de un archivo dado; la última redirige el flujo de salida, para que lo que imprime un programa se almacene en un archivo:

```
missing:~$ echo Hola > saludo.txt
missing:~$ cat saludo.txt
Hola
missing:~$ cat < saludo.txt
Hola
missing:~$ cat < saludo.txt > otro-saludo.txt
missing:~$ cat otro-saludo.txt
Hola
```

Podemos usar `>>` para agregar líneas a un archivo. Las redirecciones de entrada/salida son incluso más útiles cuando las utilizamos mediante *tuberías*. El símbolo `|` permite "encadenar" programas, de modo que la salida de uno sea la entrada de otro:

```
# obtener el contenido del directorio y mostrar sólo la última línea
missing:~$ ls -l / | tail -n1
drwxr-xr-x 1 root  root  4096 Jun 20  2019 var

# consultar la página web de google, buscar donde aparece
# su longitud en bytes y mostrar sólo la cantidad de bytes
missing:~$ curl --head --silent google.com | grep --ignore-case content-length | cut --delir
219
```

Veremos más detalles para aprovechar las tuberías en la clase sobre manipulación de datos.

Una herramienta potente y versátil

En la mayoría de los sistemas tipo Unix hay usuario que es muy importante: es el usuario llamado "root". Seguramente haya aparecido en alguno de los listados de archivos anteriores. El usuario root está por encima de (casi) todas las restricciones de acceso y puede crear, leer, actualizar y

eliminar cualquier archivo en el sistema; es muy parecido al usuario “Administrador” en Windows. Sin embargo, uno generalmente no utiliza el sistema con el nombre de usuario root, pues es demasiado fácil romper algo accidentalmente. En su lugar, utilizaremos el comando `sudo` (en Ubuntu) o bien el comando `su` (en Debian). Ambos comandos permiten ejecutar acciones que sólo están permitidas al “superusuario” o “root”. Generalmente cuando obtenemos un mensaje de eror de “permiso denegado” al ejecutar un comando, es porque hay que realizarlo con los permisos de root. Igual, ¡asegurate siempre de verificar primero qué acción estás queriendo realizar!

Una de las acciones que requiere ser usuario “root” es escribir en el sistema de archivos `sysfs` que se encuentra dentro de `/sys`. `sysfs` permite ver una serie de parámetros del núcleo del sistema o *kernel* tal como si fueran archivos, para poder reconfigurar fácilmente el sistema sobre la marcha aunque no tengamos herramientas especializadas. Por ejemplo, el archivo llamado `brightness` muestra el valor de la variable “brillo de la pantalla” de tu computadora portátil:

`/sys/class/backlight`

Podemos cambiar el brillo de la pantalla simplemente escribiendo un nuevo valor en ese archivo. Haríamos algo como lo siguiente:

```
# buscar el nombre del archivo de brillo en nuestro sistema
$ find -L /sys/class/backlight -maxdepth 2 -name '*brightness*'
/sys/class/backlight/thinkpad_screen/brightness

# movernos a dicho directorio
$ cd /sys/class/backlight/thinkpad_screen

# escribir un nuevo valor en el archivo
$ echo 3 > brightness
Ocurrió un error al redirigir al archivo 'brightness'
open: Permiso denegado
```

¡Vaya sorpresa! ¿No se supone que deberíamos poder cambiar su valor? Lo que sucede es que los usuarios “comunes” no tienen permisos para modificar configuraciones importantes del sistema. Debemos, entonces, convertirnos en “superusuario” o, como se denomina en Debian, “root” ejecutando el comando `su -` e ingresando la clave correspondiente.

Siempre que estemos en modo “superusuario”, el shell nos lo indicará anteponiendo el símbolo `#` en cada nueva línea, en vez del clásico símbolo `$`, como se ve en el siguiente ejemplo:

```
# convertirnos en usuario root
maurom@stereo:~$ su -
Contraseña: *******
root@stereo:~#
```

Entonces, volviendo al ejemplo anterior:

```
# convertirnos en usuario root
$ su -
Contraseña: *******
#

# buscar el nombre del archivo de brillo en nuestro sistema
find -L /sys/class/backlight -maxdepth 2 -name '*brightness*'
```

```
/sys/class/backlight/thinkpad_screen/brightness

# movernos a dicho directorio
cd /sys/class/backlight/thinkpad_screen

# escribir un nuevo valor en el archivo
echo 3 > brightness
```

Como el que está intentando escribir sobre el archivo `brightness` es el usuario `root`, entonces posee los permisos correctos y el comando funciona. Es posible controlar un montón de cosas del sistema mediante `/sys`, tal como el estado de las luces del teclado (la ruta podría variar):

```
# echo 1 > /sys/class/leds/input?::scrolllock/brightness
```

Para conocer con qué usuario estamos en el sistema, podemos utilizar el comando `whoami`. Y para volver a ser un usuario normal, simplemente hay que escribir el comando `exit`.

```
root@stereo:~# echo Soy un usuario con superpoderes!
Soy un usuario con superpoderes!
root@stereo:~# whoami
root
root@stereo:~# exit
cerrar sesión
maurom@stereo:~$ whoami
maurom
maurom@stereo:~$ echo Ahora soy un usuario común
Ahora soy un usuario común
```

Próximos pasos

En este punto, ya conoces bastante del intérprete de comandos como para realizar tareas básicas. Ya deberías poder moverte por el sistema para encontrar archivos interesantes y utilizar la funcionalidad básica de la mayoría de los programas. En la próxima clase, hablaremos sobre cómo realizar y automatizar tareas más complejas utilizando el shell y muchos otros programas útiles que existen.

Ejercicios

1. Creá un nuevo directorio llamado `ejercicios` en `/tmp`.
2. Buscá el programa `touch` e investigá qué hace. El programa `man` es tu amigo.
3. Utilizando `touch`, creá un nuevo archivo llamado `primera-clase` en `ejercicios`.
4. Escribí lo siguiente en ese archivo, línea tras línea:

   ```
   #!/bin/sh
   curl --head --silent https://missing.csail.mit.edu
   ```

   La primera línea puede ser complicada de escribir. Conviene saber que, en Bash, los comentarios se indican con `#`, y el símbolo `!` tiene un significado especial incluso dentro de cadenas de comillas dobles (`"`). Bash trata diferente las cadenas de comillas dobles de

las de comillas simples (`'`), pero para este ejercicio es indistinto. Consultá la página del manual de Bash sobre *entrecomillado* o quoting para obtener más información.

5. Tratá de ejecutar el archivo. Investigá por qué no funciona. Podés utilizar el comando `ls` para ello.
6. Investigá sobre el comando `chmod` .
7. Usá `chmod` para permitir que el comando `./primera-clase` se ejecute.
8. Usá `|` y `>` para armar un comando que escriba la fecha de última modificación (que aparece en la línea "last-modified") al ejecutar el archivo `primera-clase` , en un archivo nuevo llamado `ultima-modificacion.txt` dentro de tu directorio home. Todo se debería ejecutar en una sola línea.
9. Escribí un comando que lea el nivel de carga de la batería de tu portátil o la temperatura de la CPU de tu máquina de escritorio desde `/sys` . Nota: si sos usuario de macOS, tu sistema operativo no tiene sysfs, por lo que podes omitir este ejercicio.

Licencia

Acerca de Missing Semester

Original de "The Missing Semester of Your CS Education"
https://missing.csail.mit.edu/

Traducido por el equipo de Teleinformática y Redes