El entorno de línea de comandos

En esta clase repasaremos algunas formas de mejorar el trabajo en el shell. Ya estuvimos trabajando con el intérprete de comandos, pero fundamentalmente centrados en ejecutar distintos comandos. Ahora veremos cómo ejecutar varios procesos al mismo tiempo sin perder rastro de ellos, cómo pausar o detener un proceso específico y cómo hacer que un proceso se ejecute en segundo plano.

También veremos diferentes formas de mejorar el shell y otras herramientas definiendo nuevos comandos mediante "alias", y cómo personalizar las aplicaciones mediante archivos de configuración llamados *dotfiles*. Ambas estrategias nos permitirán ahorrar tiempo, por ejemplo, al utilizar configuraciones similares en todos los equipos donde trabajamos y sin tener que escribir líneas de comandos extensos.

Control de trabajos

En ciertas ocasiones necesitarás interrumpir un proceso que aún se está ejecutando, por ejemplo, si un comando tarda demasiado en completarse (como un `find` con una estructura de directorios muy grande para buscar). La mayoría de las veces, podés pulsar `Ctrl-C` y el comando se detendrá. Pero, ¿cómo funciona esto realmente y por qué a veces no alcanza para detener el proceso?

Matar un proceso

El intérprete de comandos utiliza un mecanismo de comunicación UNIX llamado *señal* para comunicar información a los procesos. Cuando un proceso recibe una señal, pausa su ejecución, atiende la señal recibida y es posible que cambie el flujo de ejecución en función de la información que llega con la señal. Por esta razón, las señales son *interrupciones de software* .

En nuestro caso, pulsar `Ctrl-C` hace que el shell envíe una señal `SIGINT` al proceso que se está ejecutando.

El siguiente es un ejemplo mínimo de un programa Python que captura `SIGINT` y lo ignora, con lo cual no es posible detenerlo. Para matar este programa deberemos enviar una señal llamada `SIGQUIT` . Esto se realiza pulsando `Ctrl-\` . El siguiente es un ejemplo mínimo de un programa Python que captura `SIGINT` y lo ignora, con lo cual no es posible detenerlo. Para matar este programa deberemos enviar una señal llamada `SIGQUIT` . Esto se realiza pulsando .

```
#!/usr/bin/env python
import signal, time

def handler(signum, time):
    print("\nHe recibido SIGINT, pero no me voy a detener")

signal.signal(signal.SIGINT, handler)
i = 0
while True:
    time.sleep(.1)
    print("\r{}".format(i), end="")
    i += 1
```

Esto es lo que sucede si enviamos `SIGINT` dos veces a este programa, seguido de `SIGQUIT` . En la terminal, el símbolo `^` suele representar una pulsación de la tecla `Ctrl` .

```
$ python sigint.py
24^C
He recibido SIGINT, pero no me voy a detener
26^C
He recibido SIGINT, pero no me voy a detener
30^\[1]    39913 quit       python sigint.py
```

Si bien las señales `SIGINT` y `SIGQUIT` están asociadas generalmente a la finalización de un proceso, una señal más genérica para solicitar que un proceso finalice "correctamente" es la señal `SIGTERM` . Para enviar esta señal podemos usar el comando `kill` , con la sintaxis `kill -TERM <PID>` .

Pausar procesos y ejecutarlos en segundo plano

Las señales pueden hacer otras cosas más allá de matar un proceso. Por ejemplo, `SIGSTOP` pone un proceso en pausa. Si presionás `Ctrl-Z` en el terminal, el intérprete de comandos enviará una señal `SIGTSTP` , abreviatura de "Terminal Stop" (es decir, la versión del terminal de `SIGSTOP`).

Para reanudar el proceso que está pausado en primer plano o en segundo plano, hay que escribir el comando `fg` o `bg` , respectivamente.

El comando `jobs` muestra todos los trabajos que aún no han finalizado en la terminal actual. Cuando sea necesario hacer referencia a un proceso en particular, podés indicarlo mediante su "process id" o *pid* (el comando `pgrep` devuelve el pid de un proceso dado su nombre). De manera más intuitiva, también es posible hacer referencia a un proceso en curso mediante el símbolo de porcentaje seguido del número de trabajo (que se muestra en los `jobs`). Para indicar que se desea operar sobre el último trabajo en segundo plano, puede utilizarse la variable de entorno `$!` .

Otro dato importante es que el sufijo `&` en una línea de comando hará que el mismo se ejecute en segundo plano, volviendo al *prompt* y permitiéndote continuar con las tareas. Hay que tener en cuenta que la salida estándar del proceso iniciado continuará siendo la terminal, por lo que es posible que aparezcan líneas de salida entre nuestros comandos (para evitarlo, redireccioná la salida hacia un archivo).

Para poner en segundo plano un programa que ya se está ejecutando, pulsá `Ctrl-Z` y luego escribí el comando `bg` . Tené en cuenta que los procesos en segundo plano siguen perteneciendo a la terminal actual, y por ello morirán si cerrás la ventana de la terminal (el shell envía la señal `SIGHUP` a cada proceso hijo). Para evitar que eso suceda, cuando lances un proceso que querés que se mantenga en fondo incluso luego de cerrar la terminal, escribí la palabra clave {a4} `nohup` {/a4} antes del comando que deseas ejecutar (el proceso `nohup` captura la señal `SIGHUP` y la ignora), o bien podes ejecutar el comando {code7}disown{/code7} para evitar que un proceso existente muera al cerrar la terminal.

En la siguiente sesión de ejemplo pueden verse algunos de estos conceptos.

```
$ sleep 1000
^Z
[1]+ 18653 Detenido  sleep 1000

$ nohup sleep 2000 &
[2] 18745
nohup: se descarta la entrada y se añade la salida a 'nohup.out'
```

```
$ jobs
[1]+  Detenido      sleep 1000
[2]-  Ejecutando    nohup sleep 2000

$ bg %1
[1]- 18653 Continuando  sleep 1000

$ jobs
[1]-  Ejecutando    sleep 1000
[2]+  Ejecutando    nohup sleep 2000

$ kill -STOP %1
[1]+ 18653 Detenido  sleep 1000

$ jobs
[1]+  Detenido      sleep 1000
[2]-  Ejecutando    nohup sleep 2000

$ kill -SIGHUP %1
[1]+ 18653 Colgar (hangup)     sleep 1000

$ jobs
[2]+  Ejecutando    nohup sleep 2000

$ kill -SIGHUP %2

$ jobs
[2]+  Ejecutando    nohup sleep 2000

$ kill %2
[2]+ 18745 Terminado  nohup sleep 2000

$ jobs
```

La señal `SIGKILL` es especial, pues no puede ser capturada por el proceso y siempre hará que el mismo muera de inmediato. Sin embargo, puede tener efectos secundarios negativos, tal como dejar en ejecución aquellos procesos que fueron iniciados por el proceso padre (procesos huérfanos).

Podés obtener más información sobre éstas y otras señales aquí o bien escribiendo `man signal` o `kill -t` .

Atajos o “alias” de comandos

Escribir comandos largos que involucran muchas banderas u opciones detalladas suele tornarse aburrido. Por ello, la mayoría de los shells admiten crear *alias* de comandos. Un alias de shell es una forma abreviada de otro comando que el intérprete reemplazará automáticamente cada vez que se escribe. Por ejemplo, un alias en bash tiene la siguiente estructura:

```
alias alias_name="command_to_alias arg1 arg2"
```

Hay que tener en cuenta que no va espacio antes ni después del signo igual `=`. `alias` es un comando de shell que toma un solo argumento.

Los alias son convenientes por múltiples razones:

```
# para hacer atajos para parámetros que se usan frecuentemente
alias ll="ls -lh"

# para acortar comandos habituales
alias gs="git status"
alias gc="git commit"
alias v="vim"

# para cuando uno suele escribir mal los comandos
alias sl=ls

# para redefinir comandos existentes
alias mv="mv -i"           # -i pregunta antes de pisar archivos
alias mkdir="mkdir -p"     # -p crea directorios padre automáticamente
alias df="df -h"           # -h muestra los tamaños en formato mb/gb

# los alias pueden incluso reutilizarse
alias la="ls -A"
alias lla="la -l"

# para evitar utilizar un alias (y forzar la ejecución del comando)
# antecederlo con el símbolo \
\ls
# o bien desactivarlo con el comando unalias
unalias la

# para conocer cómo está definido un alias, usar el comando alias
alias ll
# mostrará ll='ls -lh'
```

Hay que advertir que los alias no se mantienen automáticamente. Para que un alias sea persistente, hay que incluirlo en los archivos de inicio del shell, por ejemplo en `.bashrc` o `.zshrc`, que es lo que veremos a continuación.

Archivos de configuración o “Dotfiles”

Muchos programas se configuran utilizando archivos de texto plano conocido como *dotfiles*, debido a que los nombres de los archivos comienzan con un `.` (dot significa punto en inglés). Los archivos cuyo nombre comienza con un punto (por ejemplo `~/.vimrc`), de manera estándar no se muestran al listar un directorio con `ls`.

Los shells son un ejemplo de programas configurados con dichos archivos. Al iniciarse, tu shell leerá muchos archivos para cargar su configuración y todo el proceso puede ser bastante complejo. Aquí encontrarás un excelente recurso sobre el tema.

Para `bash`, editar tu `.bashrc` o `.bash_profile` funcionará en la mayoría de los sistemas.

Allí podés incluir comandos que querés que se ejecuten al inicio, como los alias que acabamos de describir o modificaciones a tu variable de entorno `PATH`. De hecho, muchos programas te pedirán que incluyas una línea como `export PATH="$PATH:/path/to/program/bin"` en el archivo de configuración de tu shell para que se puedan encontrar sus archivos binarios (ejecutables).

Algunos ejemplos de herramientas que se pueden configurar mediante dotfiles son:

- `bash` - `~/.bashrc`, `~/.bash_profile`
- `git` - `~/.gitconfig`
- `vim` - `~/.vimrc` y el directorio `~/.vim`
- `ssh` - `~/.ssh/config`
- `tmux` - `~/.tmux.conf`

¿Cómo debes organizar tus dotfiles? Deben estar en su propio directorio, bajo control de versiones, y **enlazados simbólicamente** a su lugar usando un script. Esto tiene como beneficios:

- **Instalación fácil**: si iniciás sesión en una nueva máquina, aplicar tus personalizaciones llevará solo un minuto.
- **Portabilidad**: tus herramientas funcionarán de la misma manera en todas partes.
- **Sincronización**: puedes actualizar tus archivos de configuración en cualquier lugar y mantenerlos todos sincronizados.
- **Seguimiento de cambios**: probablemente mantendrás tus archivos de configuración durante toda tu carrera programando, y tener un historial de versiones es bueno para proyectos de larga duración.

¿Qué deberías poner en tus dotfiles? Podés obtener información sobre la configuración de tus herramientas leyendo la documentación en línea o las páginas de manual. Otra buena manera es buscar en Internet publicaciones en blogs sobre programas específicos, donde los autores cuentan sus personalizaciones preferidas. Otra forma mas de aprender acerca de las personalizaciones es mirar los dotfiles de otras personas: podés encontrar muchísimos repositorios de dotfiles. Podés ver el más popular aquí (no obstante te recomendamos no copiar ciegamente las configuraciones). Aquí hay otro buen recurso sobre el tema.

Emuladores de terminal

Junto con la personalización de su shell, vale la pena pasar un tiempo pensando en que **emulador de terminal** elejir y su configuración. Hay muchos emuladores de terminal por ahí (aquí hay una comparación).

Dado que puedes pasar de cientos a miles de horas en tu terminal, vale la pena mirar su configuración. Algunos de los aspectos que podés modificar en tu terminal incluyen:

- Elección de la tipografía
- Esquema de colores
- Atajos de teclado
- Soporte de Pestañas / Paneles
- Configuración de desplazamiento hacia atrás
- Rendimiento (algunos terminales más nuevos como Alacritty o kitty aprovechan la aceleración por GPU).

Ejercicios

Control de trabajos

1. Por lo que hemos visto, podemos usar los comandos `ps aux | grep` para obtener los pids de nuestros trabajos y luego matarlos, pero hay mejores maneras de hacerlo. Inicie un trabajo `sleep 10000` en una terminal, envíalo a segundo plano con `Ctrl-Z` y continúa su ejecución con `bg`. Ahora usa `pgrep` para encontrar su pid y `pkill` para matarlo sin tener que escribir el pid en sí. (Sugerencia: use los parámetros `-af`).

2. Digamos que no deseas comenzar un proceso hasta que se complete otro, ¿cómo lo harías? En este ejercicio, nuestro proceso limitante será `sleep 60 &`. Una forma de lograr esto es usar el comando `wait`. Intenta iniciar el comando sleep y `ls`, pero haciendo que este espere hasta que finalice el proceso en segundo plano.

   Sin embargo, esta estrategia no funcionará si comenzamos en una sesión bash diferente, ya que `wait` solo funciona para procesos hijos. Una característica que no discutimos en las notas es que el estado de salida del comando `kill` será cero en caso de éxito, y distinto de cero en caso contrario. `kill -0` no envía una señal pero tendrá un estado de salida distinto de cero si el proceso no existe. Escribe una función bash llamada `pidwait` que tome un pid y espere hasta que dicho proceso se complete. Deberás usar `sleep` para evitar desperdiciar CPU innecesariamente

Atajos o Alias

1. Creá un alias `dc` que ejecute `cd` para cuando lo escribís incorrectamente.

2. Ejecutá `history | awk '{$1="";print substr($0,2)}' | sort | uniq -c | sort -n | tail -n 10` para obtener tus 10 comandos más utilizados y considerá escribir alias más cortos para ellos. Nota: esto funciona para Bash; si usás ZSH, utilizá `history 1` en lugar de solo `history`.

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