36

Exotica
Exotica

In this, the final chapter of our journey, we will look at some odds and ends. While we have certainly covered a lot of ground in the previous chapters, there are many bash features that we have not covered. Most are fairly obscure and useful mainly to those integrating bash into a Linux distribution. However, there are a few that, while not in common use, are helpful for certain pro gramming problems. We will cover them here.
En este, el capítulo final de nuestro viaje, veremos algunos extremos. Aunque ciertamente hemos cubierto mucho terreno en los capítulos anteriores, hay muchas características de bash que no hemos cubierto. La mayoría son bastante oscuros y útiles principalmente para aquellos que integran bash en una distribución de Linux. Sin embargo, hay algunos que, aunque no son de uso común, son útiles para ciertos problemas de programación. Los cubriremos aquí.

Group Commands and Subshells
Comandos de grupo y subcapas

bash allows commands to be grouped together. This can be done in one of two ways, either with a group command or with a subshell.
bash permite que los comandos se agrupen. Esto se puede hacer de dos formas, ya sea con un comando de grupo o con un subshell.

Here is the syntax of a group command:
Aquí está la sintaxis de un comando de grupo:

```
{ command1; command2; [command3; ...] }
```

Here is the syntax of a subshell:
Aquí está la sintaxis de una subcapa:

```
(command1; command2; [command3;...])
```

The two forms differ in that a group command surrounds its commands with braces and a subshell uses parentheses. It is important to note that because of the way bash implements group commands, the braces must be separated from the commands by a space and the last command must be terminated with either a semicolon or a newline prior to the closing brace. So, what are group commands and subshells good for? While they have an important difference (which we will get to in a moment), they are both used to manage redirection. Let's consider a script segment that performs redirections on multiple commands.
Las dos formas se diferencian en que un comando de grupo rodea sus comandos con llaves y una subcapa usa paréntesis. Es importante notar que debido a la forma en que bash implementa los comandos de grupo, las llaves deben estar separadas de los comandos por un espacio y el último comando debe terminar con un punto y coma o una nueva línea antes de la llave de cierre. Entonces, ¿para qué sirven los comandos de grupo y las subcapas? Si bien tienen una diferencia importante (a la que llegaremos en un momento), ambos se utilizan para administrar la redirección. Consideremos un segmento de secuencia de comandos que realiza redirecciones en varios comandos.

```
ls -l > output.txt
echo "Listing of foo.txt" >> output.txt
cat foo.txt >> output.txt
```

This is pretty straightforward. Three commands have their output redirected to a file named output.txt. Using a group command, we could code this as follows:
Esto es bastante sencillo. Tres comandos tienen su salida redirigida a un archivo llamado output.txt. Usando un comando de grupo, podríamos codificar esto de la siguiente manera:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

Using a subshell is similar.
Usar una subcapa es similar.

```
(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

Using this technique we have saved ourselves some typing, but where a group command or subshell really shines is with pipelines. When construct ing a pipeline of commands, it is often useful to combine the results of several commands into a single stream. Group commands and subshells make this easy.
Usando esta técnica, nos hemos ahorrado algo de escritura, pero donde un comando de grupo o subshell realmente brilla es con pipelines. Al construir una canalización de comandos, a menudo es útil combinar los resultados de varios comandos en un solo flujo. Los comandos de grupo y las subcapas lo hacen fácil.

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

Here we have combined the output of our three commands and piped them into the input of lpr to produce a printed report.
Aquí hemos combinado la salida de nuestros tres comandos y los hemos conectado a la entrada de lpr para producir un informe impreso.

In the script that follows, we will use groups commands and look at several programming techniques that can be employed in conjunction with associative arrays. This script, called array-2 , when given the name of a directory, prints a listing of the files in the directory along with the names of the file's owner and group owner. At the end of the listing, the script prints a tally of the number of files belonging to each owner and group. Here we see the results (condensed for brevity) when the script is given the directory /usr/bin:
En el script que sigue, usaremos comandos de grupos y veremos varias técnicas de programación que pueden emplearse junto con matrices asociativas. Este script, llamado array-2, cuando se le da el nombre de un directorio, imprime una lista de los archivos en el directorio junto con los nombres del propietario del archivo y del propietario del grupo. Al final de la lista, el guión imprime un recuento de la cantidad de archivos que pertenecen a cada propietario y grupo. Aquí vemos los resultados (condensados por brevedad) cuando el script recibe el directorio /usr/bin:

```
[me@linuxbox ~]$ array-2 /usr/bin
/usr/bin/2to3-2.6   root        root

.....
```

Here is a listing (with line numbers) of the script:
Aquí hay una lista (con números de línea) del script:

```
#!/bin/bash

# array-2: Use arrays to tally file owners

declare -A files file_group file_owner groups owners

if [[ ! -d "$1" ]]; then
    echo "Usage: array-2 dir" >&2
    exit 1
fi

for i in "$1"/*; do
    owner="$(stat -c %U "$i")"
    group="$(stat -c %G "$i")"
    files["$i"]="$i"
    file_owner["$i"]="$owner"
```

```
    file_group["$i"]="$group"
    ((++owners[$owner]))
    ((++groups[$group]))
done

# List the collected files
{ for i in "${files[@]}"; do
    printf "%-40s %-10s %-10s\n" \
        "$i" "${file_owner["$i"]}" "${file_group["$i"]}"
done } | sort
echo

# List owners
echo "File owners:"
{ for i in "${!owners[@]}"; do
    printf "%-10s: %5d file(s)\n" "$i" "${owners["$i"]}"
done } | sort
echo

# List groups
echo "File group owners:"
{ for i in "${!groups[@]}"; do
    printf "%-10s: %5d file(s)\n" "$i" "${groups["$i"]}"
done } | sort
```

Let's take a look at the mechanics of this script.
Echemos un vistazo a la mecánica de este script.

Line 5: Associative arrays must be created with the declare command using the -A option. In this script, we create five arrays as follows:
Línea 5: Las matrices asociativas deben crearse con el comando declare usando la opción -A. En este script, creamos cinco matrices de la siguiente manera:

- files contains the names of the files in the directory, indexed by filename.
  files contiene los nombres de los archivos en el directorio, indexados por nombre de archivo.
- file_group contains the group owner of each file, indexed by filename.
  file_group contiene el propietario del grupo de cada archivo, indexado por nombre de archivo.
- file_owner contains the owner of each file, indexed by filename.
  file_owner contiene el propietario de cada archivo, indexado por nombre de archivo.
- groups contains the number of files belonging to the indexed group.
  grupos contiene el número de archivos que pertenecen al grupo indexado.
- owners contains the number of files belonging to the indexed owner.
  propietarios contiene el número de archivos que pertenecen al propietario indexado.

Lines 7–10: These lines check to see that a valid directory name was passed as a positional parameter. If not, a usage message is displayed, and the script exits with an exit status of 1.
Líneas 7-10: estas líneas comprueban que se haya pasado un nombre de directorio válido como parámetro posicional. De lo contrario, se muestra un mensaje de uso y el script finaliza con un estado de salida de 1.

Lines 12–20: These lines loop through the files in the directory. Using the stat command, lines 13 and 14 extract the names of the file owner and group owner and assign the values to their respective arrays (lines 16 and 17) using the name of the file as the array index. Likewise, the filename itself is assigned to the files array (line 15).
Líneas 12-20: estas líneas recorren los archivos del directorio. Usando el comando stat, las líneas 13 y 14 extraen los nombres del propietario del archivo y del propietario del grupo y asignan los valores a sus respectivas matrices (líneas 16 y 17) usando el nombre del archivo como índice de la matriz. Asimismo, el nombre del archivo en sí se asigna a la matriz de archivos (línea 15).

Lines 18–19: The total number of files belonging to the file owner and group owner are incremented by one.
Líneas 18-19: el número total de archivos que pertenecen al propietario del archivo y al propietario del grupo se incrementa en uno.

Lines 22–27: The list of files is output. This is done using the "${array[@]}" parameter expansion, which expands into the entire list of array ele ments with each element treated as a separate word. This allows for the possibility that a filename may contain embedded spaces. Also note that the entire loop is enclosed in braces thus forming a group command. This permits the entire output of the loop to be piped into the sort command. This is necessary because the expansion of the array elements is not sorted.
Líneas 22–27: Se emite la lista de archivos. Esto se hace usando la expansión del parámetro "$ {array [@]}", que se expande a la lista completa de elementos del array con cada elemento tratado como una palabra separada. Esto permite la posibilidad de que un nombre de archivo contenga espacios incrustados. También tenga en cuenta que todo el bucle está encerrado entre llaves, formando así un comando de grupo. Esto permite que toda la salida del bucle se canalice al comando sort. Esto es necesario porque la expansión de los elementos de la matriz no está ordenada.

Lines 29–40: These two loops are similar to the file list loop except that they use the "${!array[@]}" expansion, which expands into the list of array indexes rather than the list of array elements.
Líneas 29-40: estos dos bucles son similares al bucle de lista de archivos, excepto que utilizan la expansión "${!Array[@]}", que se expande en la lista de índices de matriz en lugar de en la lista de elementos de matriz.

Process Substitution
Sustitución de procesos

While they look similar and can both be used to combine streams for redi rection, there is an important difference between group commands and subshells. Whereas a group command executes all of its commands in the current shell, a subshell (as the name suggests) executes its commands in a child copy of the current shell. This means the environment is copied and given to a new instance of the shell. When the subshell exits, the copy of the environment is lost, so any changes made to the subshell's environment (including variable assignment) are lost as well. Therefore, in most cases, unless a script requires a subshell, group commands are preferable to sub shells. Group commands are both faster and require less memory.
Si bien tienen un aspecto similar y ambos se pueden usar para combinar secuencias para redirigir, existe una diferencia importante entre los comandos de grupo y las subcapas. Mientras que un comando de grupo ejecuta todos sus comandos en el shell actual, un subshell (como sugiere el nombre) ejecuta sus comandos en una copia secundaria del shell actual. Esto significa que el entorno se copia y se asigna a una nueva

instancia del shell. Cuando la subcapa sale, la copia del entorno se pierde, por lo que cualquier cambio realizado en el entorno de la subcapa (incluida la asignación de variables) también se pierde. Por lo tanto, en la mayoría de los casos, a menos que un script requiera una subcapa, los comandos de grupo son preferibles a las subcapa. Los comandos de grupo son más rápidos y requieren menos memoria

We saw an example of the subshell environment problem in Chapter 28, when we discovered that a read command in a pipeline does not work as we might intuitively expect. To recap, if we construct a pipeline like this:
Vimos un ejemplo del problema del entorno de subcapa en el Capítulo 28, cuando descubrimos que un comando de lectura en una canalización no funciona como podríamos esperar intuitivamente. En resumen, si construimos una canalización como esta:

```
echo "foo" | read
echo $REPLY
```

the content of the REPLY variable is always empty because the read command is executed in a subshell, and its copy of REPLY is destroyed when the subshell terminates.
el contenido de la variable REPLY siempre está vacío porque el comando de lectura se ejecuta en una subcapa y su copia de REPLY se destruye cuando termina la subcapa.

Because commands in pipelines are always executed in subshells, any command that assigns variables will encounter this issue. Fortunately, the shell provides an exotic form of expansion called process substitution that can be used to work around this problem.
Debido a que los comandos en las canalizaciones siempre se ejecutan en subcapas, cualquier comando que asigne variables encontrará este problema. Afortunadamente, el shell proporciona una forma exótica de expansión denominada sustitución de procesos que se puede utilizar para solucionar este problema.

Process substitution is expressed in two ways.
La sustitución de procesos se expresa de dos formas.

For processes that produce standard output, it looks like this:
Para los procesos que producen una salida estándar, se ve así:

```
<(list)
```

For processes that intake standard input, it looks like this:
Para los procesos que toman entrada estándar, se ve así:

```
>(list)
```

where list is a list of commands.
donde lista es una lista de comandos.

To solve our problem with read , we can employ process substitution like this.
Para resolver nuestro problema con la lectura, podemos emplear la sustitución de procesos como esta.

```
read < <(echo "foo")
echo $REPLY
```

Process substitution allows us to treat the output of a subshell as an ordinary file for purposes of redirection. In fact, since it is a form of expansion, we can examine its real value.
La sustitución de procesos nos permite tratar la salida de una subcapa como un archivo ordinario para fines de redirección. De hecho, dado que es una forma de expansión, podemos examinar su valor real.

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

By using echo to view the result of the expansion, we see that the output of the subshell is being provided by a file named /dev/fd/63.
Al usar echo para ver el resultado de la expansión, vemos que la salida de la subcapa la proporciona un archivo llamado /dev/fd/63.

Process substitution is often used with loops containing read . Here is an example of a read loop that processes the contents of a directory listing created by a subshell:
La sustitución de procesos se usa a menudo con bucles que contienen read. A continuación, se muestra un ejemplo de un bucle de lectura que procesa el contenido de una lista de directorios creada por una subcapa:

```
#!/bin/bash

# pro-sub: demo of process substitution

while read attr links owner group size date time filename; do
    cat << EOF
        Filename:     $filename
        Size:         $size
        Owner:        $owner
        Group:        $group
        Modified:     $date $time
        Links:        $links
        Attributes:   $attr
EOF
done < <(ls -l | tail -n +2)
```

The loop executes read for each line of a directory listing. The listing itself is produced on the final line of the script. This line redirects the out put of the process substitution into the standard input of the loop. The tail command is included in the process substitution pipeline to eliminate the first line of the listing, which is not needed.
El bucle ejecuta read para cada línea de una lista de directorios. El listado en sí se produce en la última línea

del guión. Esta línea redirige la salida de la sustitución del proceso a la entrada estándar del bucle. El comando tail se incluye en la canalización de sustitución del proceso para eliminar la primera línea del listado, que no es necesaria.

When executed, the script produces output like this:
Cuando se ejecuta, el script produce un resultado como este:

```
[me@linuxbox ~]$ pro-sub | head -n 20
Filename: addresses.ldif
Size: 14540
Owner: me
Group: me
Modified: 2009-04-02 11:12
Links: 1
Attributes: -rw-r--r—

......
```

Traps
Trampas

In Chapter 10, we saw how programs can respond to signals. We can add this capability to our scripts, too. While the scripts we have written so far have not needed this capability (because they have very short execution times and do not create temporary files), larger and more complicated scripts may benefit from having a signal handling routine.
En el Capítulo 10, vimos cómo los programas pueden responder a las señales. También podemos agregar esta capacidad a nuestros scripts. Si bien los scripts que hemos escrito hasta ahora no han necesitado esta capacidad (porque tienen tiempos de ejecución muy cortos y no crean archivos temporales), los scripts más grandes y complicados pueden beneficiarse de tener una rutina de manejo de señales.

When we design a large, complicated script, it is important to consider what happens if the user logs off or shuts down the computer while the script is running. When such an event occurs, a signal will be sent to all affected processes. In turn, the programs representing those processes can perform actions to ensure a proper and orderly termination of the program. Let's say, for example, that we wrote a script that created a temporary file during its execution. In the course of good design, we would have the script delete the file when the script finishes its work. It would also be smart to have the script delete the file if a signal is received indicating that the program was going to be terminated prematurely.
Cuando diseñamos un script grande y complicado, es importante considerar qué sucede si el usuario cierra la sesión o apaga la computadora mientras se ejecuta el script. Cuando ocurra tal evento, se enviará una señal a todos los procesos afectados. A su vez, los programas que representan esos procesos pueden realizar acciones para asegurar una terminación adecuada y ordenada del programa. Digamos, por ejemplo, que escribimos un script que creaba un archivo temporal durante su ejecución. En el curso de un buen diseño, haríamos que el script elimine el archivo cuando el script finalice su trabajo. También sería

inteligente que el script elimine el archivo si se recibe una señal que indique que el programa se cerrará antes de tiempo.

bash provides a mechanism for this purpose known as a trap. Traps are implemented with the appropriately named builtin command, trap . trap uses the following syntax:
bash proporciona un mecanismo para este propósito conocido como trampa. Las trampas se implementan con el comando incorporado con el nombre apropiado, trap. trap utiliza la siguiente sintaxis:

```
trap argument signal [signal...]
```

---

where argument is a string that will be read and treated as a command and signal is the specification of a signal that will trigger the execution of the interpreted command.
donde argumento es una cadena que se leerá y tratará como un comando y la señal es la especificación de una señal que activará la ejecución del comando interpretado.

Here is a simple example:
Aquí hay un ejemplo simple:

```
#!/bin/bash

# trap-demo: simple signal handling demo

trap "echo 'I am ignoring you.'" SIGINT SIGTERM
for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

This script defines a trap that will execute an echo command each time either the SIGINT or SIGTERM signal is received while the script is running.
Este script define una trampa que ejecutará un comando de eco cada vez que se reciba la señal SIGINT o SIGTERM mientras se ejecuta el script.

Execution of the program looks like this when the user attempts to stop the script by pressing ctrl -C:
La ejecución del programa se ve así cuando el usuario intenta detener el script presionando ctrl -C:

```
[me@linuxbox ~]$ trap-demo
Iteration 1 of 5
Iteration 2 of 5
^CI am ignoring you.
Iteration 3 of 5
^CI am ignoring you.
Iteration 4 of 5
Iteration 5 of 5
```

As we can see, each time the user attempts to interrupt the program, the message is printed instead.
Como podemos ver, cada vez que el usuario intenta interrumpir el programa, en su lugar se imprime el mensaje.

Constructing a string to form a useful sequence of commands can be awkward, so it is common practice to specify a shell function as the com mand. In this example, a separate shell function is specified for each signal to be handled:
La construcción de una cadena para formar una secuencia útil de comandos puede ser incómoda, por lo que es una práctica común especificar una función de shell como el comando. En este ejemplo, se especifica una función de shell separada para cada señal a manejar:

```
#!/bin/bash

# trap-demo2: simple signal handling demo

exit_on_signal_SIGINT () {
    echo "Script interrupted." 2>&1
    exit 0
}

exit_on_signal_SIGTERM () {
    echo "Script terminated." 2>&1
    exit 0
}

trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

This script features two trap commands, one for each signal. Each trap, in turn, specifies a shell function to be executed when the particular signal is received. Note the inclusion of an exit command in each of the signalhandling functions. Without an exit , the script would continue after com pleting the function.
Este script presenta dos comandos de trampa, uno para cada señal. Cada trampa, a su vez, especifica una función de shell que se ejecutará cuando se reciba la señal particular. Tenga en cuenta la inclusión de un comando de salida en cada una de las funciones de manejo de señales. Sin una salida, el script continuaría después de completar la función.

When the user presses ctrl -C during the execution of this script, the results look like this:
Cuando el usuario presiona ctrl -C durante la ejecución de este script, los resultados se ven así:

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5
^CScript interrupted.
```

Temporary Files
Archivos temporales

One reason signal handlers are included in scripts is to remove temporary files that the script may create to hold intermediate results during execution. There is something of an art to naming temporary files.
One reason signal handlers are included in scripts is to remove temporary files that the script may create to hold intermediate results during execution. There is something of an art to naming temporary files.

Traditionally, programs on Unixlike systems create their temporary files in the /tmp directory, a shared directory intended for such files. However, since the directory is shared, this poses certain security concerns, particularly for programs running with superuser privileges.
Tradicionalmente, los programas en sistemas tipo Unix crean sus archivos temporales en el directorio / tmp, un directorio compartido destinado a dichos archivos. Sin embargo, dado que el directorio es compartido, esto plantea ciertos problemas de seguridad, particularmente para los programas que se ejecutan con privilegios de superusuario.

Aside from the obvious step of setting proper permissions for files exposed to all users of the system, it is important to give temporary files nonpredictable filenames. This avoids an exploit known as a temp race attack. One way to create a nonpredictable (but still descriptive) name is to do something like this:
Aparte del paso obvio de establecer los permisos adecuados para los archivos expuestos a todos los usuarios del sistema, es importante dar a los archivos temporales nombres de archivo no predecibles. Esto evita un exploit conocido como ataque de carrera temporal. Una forma de crear un nombre no predecible (pero aún descriptivo) es hacer algo como esto:

```
tempfile=/tmp/$(basename $0).$$.$RANDOM
```

This will create a filename consisting of the program's name, followed by its process ID (PID), followed by a random integer. Note, however, that the $RANDOM shell variable returns a value only in the range of 1–32767, which is not a large range in computer terms, so a single instance of the variable is not sufficient to overcome a determined attacker.
Esto creará un nombre de archivo que consiste en el nombre del programa, seguido de su ID de proceso (PID), seguido de un número entero aleatorio. Sin embargo, tenga en cuenta que la variable de shell $ RANDOM devuelve un valor solo en el rango de 1–32767, que no es un rango grande en términos informáticos, por lo que una sola instancia de la variable no es suficiente para vencer a un atacante determinado.

A better way is to use the mktemp program (not to be confused with the mktemp standard library function) to both name and create the temporary file.
Una mejor manera es usar el programa mktemp (que no debe confundirse con la función de biblioteca estándar de mktemp) para nombrar y crear el archivo temporal.

The mktemp program accepts a template as an argument that is used to build the filename. The template should include a series of X characters, which are replaced by a corresponding number of random letters and numbers. The longer the series of X characters, the longer the series of random characters.
El programa mktemp acepta una plantilla como argumento que se utiliza para construir el nombre del archivo. La plantilla debe incluir una serie de caracteres X, que se reemplazan por un número correspondiente de letras y números aleatorios. Cuanto más larga sea la serie de X caracteres, más larga será la serie de caracteres aleatorios.

Here is an example:
Aquí hay un ejemplo:

```
tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
```

This creates a temporary file and assigns its name to the variable tempfile. The X characters in the template are replaced with random letters and numbers so that the final filename (which, in this example, also includes the expanded value of the special parameter $$ to obtain the PID) might be something like this:
Esto crea un archivo temporal y asigna su nombre a la variable tempfile. Los caracteres X en la plantilla se reemplazan con letras y números aleatorios para que el nombre de archivo final (que, en este ejemplo, también incluye el valor expandido del parámetro especial $$ para obtener el PID) podría ser algo como esto:

```
/tmp/foobar.6593.UOZuvM6654
```

For scripts that are executed by regular users, it may be wise to avoid the use of the /tmp directory and create a directory for temporary files within the user's home directory, with a line of code such as this:
Para los scripts que son ejecutados por usuarios regulares, puede ser conveniente evitar el uso del directorio / tmp y crear un directorio para archivos temporales dentro del directorio de inicio del usuario, con una línea de código como esta:

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```

Asynchronous Execution with wait
Ejecución asincrónica con espera

It is sometimes desirable to perform more than one task at the same time.
A veces es deseable realizar más de una tarea al mismo tiempo.

We have seen how all modern operating systems are at least multitasking if not multiuser as well. Scripts can be constructed to behave in a multitasking fashion.
Hemos visto cómo todos los sistemas operativos modernos son al menos multitarea, si no multiusuario también. Los scripts se pueden construir para comportarse de manera multitarea.

Usually this involves launching a script that, in turn, launches one or more child scripts to perform an additional task while the parent script continues to run. However, when a series of scripts runs this way, there can be problems keeping the parent and child coordinated. That is, what if the parent or child is dependent on the other and one script must wait for the other to finish its task before finishing its own?
Por lo general, esto implica el lanzamiento de un script que, a su vez, inicia uno o más scripts secundarios para realizar una tarea adicional mientras el script principal continúa ejecutándose. Sin embargo, cuando

una serie de secuencias de comandos se ejecuta de esta manera, puede haber problemas para mantener la coordinación del padre y el hijo. Es decir, ¿qué pasa si el padre o el hijo dependen del otro y un script debe esperar a que el otro termine su tarea antes de terminar la suya propia?

bash has a builtin command to help manage asynchronous execution such as this. The wait command causes a parent script to pause until a specified process (i.e., the child script) finishes. To demonstrate this, we will need two scripts. The first is a parent script.
bash tiene un comando incorporado para ayudar a administrar la ejecución asincrónica como esta. El comando de espera hace que una secuencia de comandos principal se detenga hasta que finalice un proceso específico (es decir, la secuencia de comandos secundaria). Para demostrar esto, necesitaremos dos scripts. El primero es un script padre.

```
#!/bin/bash

# async-parent: Asynchronous execution demo (parent)

echo "Parent: starting..."
echo "Parent: launching child script..."

async-child &
pid=$!
echo "Parent: child (PID= $pid) launched."

echo "Parent: continuing..."
sleep 2
echo "Parent: pausing to wait for child to finish..."
wait "$pid"

echo "Parent: child is finished. Continuing..."
echo "Parent: parent is done. Exiting."
```

This second is a child script.
Este segundo es un guión secundario.

```
#!/bin/bash
# async-child: Asynchronous execution demo (child)

echo "Child: child is running..."
sleep 5
echo "Child: child is done. Exiting."
```

In this example, we see that the child script is simple. The real action is being performed by the parent. In the parent script, the child script is launched and put into the background. The process ID of the child script is recorded by assigning the pid variable with the value of the $! shell parameter, which will always contain

the process ID of the last job put into the background.
En este ejemplo, vemos que el script secundario es simple. La acción real la realiza el padre. En el script principal, el script secundario se inicia y se coloca en segundo plano. El ID de proceso del script secundario se registra asignando la variable pid con el valor de $! parámetro de shell, que siempre contendrá el ID de proceso del último trabajo puesto en segundo plano.

The parent script continues and then executes a wait command with the PID of the child process. This causes the parent script to pause until the child script exits, at which point the parent script concludes.
El script padre continúa y luego ejecuta un comando de espera con el PID del proceso hijo. Esto hace que la secuencia de comandos principal se detenga hasta que se cierre la secuencia de comandos secundaria, momento en el que concluye la secuencia de comandos principal.

When executed, the parent and child scripts produce the following output:
Cuando se ejecutan, los scripts padre e hijo producen el siguiente resultado:

```
[me@linuxbox ~]$ async-parent
Parent: starting...
Parent: launching child script...
Parent: child (PID= 6741) launched.
Parent: continuing...
Child: child is running...
Parent: pausing to wait for child to finish...
Child: child is done. Exiting.
Parent: child is finished. Continuing...
Parent: parent is done. Exiting.
```

Named Pipes
Tubos con nombre

In most Unix-like systems, it is possible to create a special type of file called a named pipe. Named pipes are used to create a connection between two processes and can be used just like other types of files. They are not that popular, but they’re good to know about.
En la mayoría de los sistemas similares a Unix, es posible crear un tipo especial de archivo llamado canalización con nombre. Las canalizaciones con nombre se utilizan para crear una conexión entre dos procesos y se pueden utilizar como otros tipos de archivos. No son tan populares, pero es bueno conocerlos.

There is a common programming architecture called client-server, which can make use of a communication method such as named pipes, as well as other kinds of interprocess communication such as network connections.
Existe una arquitectura de programación común llamada cliente-servidor, que puede hacer uso de un método de comunicación como canalizaciones con nombre, así como otros tipos de comunicación entre procesos, como conexiones de red.

The most widely used type of client-server system is, of course, a web browser communicating with a web server. The web browser acts as the client, making requests to the server, and the server responds to the browser with web pages.
El tipo de sistema cliente-servidor más utilizado es, por supuesto, un navegador web que se comunica con un servidor web. El navegador web actúa como cliente, realiza solicitudes al servidor y el servidor responde al navegador con páginas web.

Named pipes behave like files but actually form first-in first-out (FIFO) buffers. As with ordinary (unnamed) pipes, data goes in one end and emerges out the other. With named pipes, it is possible to set up something like this:
Las canalizaciones con nombre se comportan como archivos, pero en realidad forman búferes primero en entrar, primero en salir (FIFO). Al igual que con las tuberías ordinarias (sin nombre), los datos van por un extremo y emergen por el otro. Con canalizaciones con nombre, es posible configurar algo como esto:

```
process1 > named_pipe
```

and this:
y esto:

```
process2 < named_pipe
```

and it will behave like this:
y se comportará como esto:

```
process1 | process2
```

Setting Up a Named Pipe
Configurar una tubería con nombre

First, we must create a named pipe. This is done using the mkfifo command.
Primero, debemos crear una tubería con nombre. Esto se hace usando el comando mkfifo.

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me me  2018-07-17 06:41 pipe1
```

Here we use mkfifo to create a named pipe called pipe1 . Using ls , we examine the file and see that the first letter in the attributes field is p , indicating that it is a named pipe.
Aquí usamos mkfifo para crear una tubería con nombre llamada pipe1. Usando ls, examinamos el archivo y vemos que la primera letra en el campo de atributos es p, lo que indica que es una tubería con nombre.

Using Named Pipes
Usar tuberías con nombre

To demonstrate how the named pipe works, we will need two terminal win dows (or alternately, two virtual consoles). In the first terminal, we enter a simple command and redirect its output to the named pipe.
Para demostrar cómo funciona la tubería nombrada, necesitaremos dos ventanas de terminal (o alternativamente, dos consolas virtuales). En la primera terminal, ingresamos un comando simple y redirigimos su salida a la tubería nombrada.

```
[me@linuxbox ~]$ ls -l > pipe1
```

After we press enter , the command will appear to hang. This is because there is nothing receiving data from the other end of the pipe yet. When this occurs, it is said that the pipe is blocked. This condition will clear once we attach a process to the other end and it begins to read input from the pipe. Using the second terminal window, we enter this command:
Después de presionar enter, el comando parecerá colgarse. Esto se debe a que todavía no hay nada que reciba datos del otro extremo de la tubería. Cuando esto ocurre, se dice que la tubería está bloqueada. Esta condición desaparecerá una vez que adjuntemos un proceso al otro extremo y comience a leer la entrada de la tubería. Usando la segunda ventana de terminal, ingresamos este comando:

```
[me@linuxbox ~]$ cat < pipe1
```

The directory listing produced from the first terminal window appears in the second terminal as the output from the cat command. The ls command in the first terminal successfully completes once it is no longer blocked.
La lista de directorios producida desde la primera ventana de terminal aparece en la segunda terminal como la salida del comando cat. El comando ls en la primera terminal se completa con éxito una vez que ya no está bloqueado.

Summing Up
Resumiendo

Well, we have completed our journey. The only thing left to do now is practice, practice, practice. Even though we covered a lot of ground in our trek, we barely scratched the surface as far as the command line goes.
Bueno, hemos completado nuestro viaje. Lo único que queda por hacer ahora es practicar, practicar, practicar. A pesar de que cubrimos mucho terreno en nuestra caminata, apenas arañamos la superficie hasta donde llega la línea de comando.

There are still thousands of command line programs left to be discovered and enjoyed. Start digging around in /usr/bin and you'll see!
Todavía quedan miles de programas de línea de comandos por descubrir y disfrutar. ¡Empiece a buscar en /usr/bin y lo verá!