# Entrada/Salida en la shell de GNU/Linux

Autor: Rodrigo García (https://rmgss.net/contacto) 

## Brevemente sobre file-descriptors

Un programa o comando en GNU/Linux puede hacer muchas cosas y entre estas esta interactuar con otros programas o usuarios extrayendo información de algún lugar y también poner los resultados de lo que hace en otro lugar.

Típicamente el "lugar" desde donde los programas extraen información es un *stream* al que se le llama la entrada estándar o *stdin* (que viene de *standard input*). El lugar hacia donde reportan sus acciones es la salida estándar o *stdout* y donde reportan errores se llama el error estándar *stderr*. Los *streams* que utilizan los procesos son listados en *file-descriptors* asociados a cada proceso.

Estos tres streams son los más utilizados y estandarizados pero en este sistema operativo se pueden usar otros mas y podríamos utilizarlos como nos convega.

Un descriptor de archivo o *file-descriptor* es como un indicador que dice los *streams* que usa un archivo. En sistemas UNIX todo es un archivo, y al estar GNU/Linux basado en UNIX, un proceso tienen una serie de archivos asociados a este que lo describen y ayudan a gestionarlo. (en este enlace puedes ver otra descripción sencilla sobre *file-descriptors*)

Por ejemplo, podemos ver los archivos asociados un proceso, en este caso el proceso de la terminal que tenemos abierta con:

```
lsof -p $BASHPID
```

Donde `lsof` es un comando que se utiliza para ver los archivos abiertos y con `-p $BASHPID` filtramos solo los archivos del identificador de proceso (`PID`) de la misma terminal abierta, en bash la variable `$BASHPID` es la que almacena el PID de la terminal abierta, para filtrar solamente los *file-descriptors* asociados al proceso de la terminal actual podemos usar:

```
lsof +f g -ap $BASHPID -d 0,1,2
```

(Ejemplo extraído de http://wiki.bash-hackers.org/howto/redirection_tutorial)

El resultado muestra entre muchas cosas los file descriptros (FD) usados por el proceso con PID 6841:

```
COMMAND  PID   USER   FD   TYPE FILE-FLAG DEVICE SIZE/OFF NODE NAME
bash    6481 lorilu    0u   CHR        RW  136,5      0t0    8 /dev/pts/5
bash    6481 lorilu    1u   CHR        RW  136,5      0t0    8 /dev/pts/5
bash    6481 lorilu    2u   CHR        RW  136,5      0t0    8 /dev/pts/5
```

En el caso anterior con `-d 0,1,2` filtramos los file descriptors 0,1,2 que se asocian a *stdin*, *stdout* y *stderr* respectivamente ya que:

```
         Stream                File-descriptor (FD)

 stdin (entrada estandar) --->      0
 stdout (salida estandar) --->      1
 stderr (error estandar)  --->      2
```

Si queremos ver todos los file descriptors asociados a un proceso, podemos listar el contenido del directorio `/proc/<PID>/fd`, donde `<PID>` viene a ser el PID del proceso.

Los operadores de redirección, entre los que están `> >> < n>m &>` o las tuberías `|` ayudan a manipular la fuente o destino de información con la que los programas interactúan y se crearon para proporcionar un alto nivel de flexiblidad.

## Sobre operadores de redirección

Ya sabemos que los programas actúan sobre *streams* definidos por *file-descriptors,* ahora podemos utilizar los operadores de redirección, que están brevemente descritos en http://www.catonmat.net/download/bash-redirections-cheat-sheet.pdf

### >

- `comando > archivo`

El operador > redirije el *stdout* y lo guarda en un archivo.

```
# guarda la salida estandar de cada comando en archivos nuevos
ls > lista.txt
ps -aux --forest > lista_procesos.txt
```

Al usar `comando > archivo` se debe tener en cuenta que si `archivo` ya existe su contenido sera **reemplazado** por la salida estándar de `comando`.

## >>

- `comando >> archivo`

>> al igual que > redirije el *stdout*, pero a diferencia de este adiciona el contenido de stdout al archivo y no lo reemplaza.

## &> y &>>

- `comando &> archivo`
- `comando &>> archivo`

Por defecto redirije la *stdout* y *stderr* al archivo, pero puede usarse tambien de otros *file-descriptors* que veremos mas adelante.

## <

- `comando < archivo`

Redirecciona el contenido del archivo a la *stdin*, si el comando lee la entrada estandar estara en realidad leyendo el contenido del archivo y lo usara como argumento.

## |

- `comando1 | comando2`

Redirije *stdout* del comando1 a *stdin* y si el comando2 la lee la usara como argumento.

## n>

- `comando 2> archivo`

Funciona igual que > solo que en este caso 2> redirije el *file-descriptor* 2 que es *stderr* al archivo, de hecho se puede usar cualquier file desriptor por ejemplo `comando 11> archivo` que usara el *file-descriptor* 11 y lo guardara en archivo. El comportamiento es el mismo con n>>.

## n<

- `comando 1< archivo`

Al igual que < redirije el contenido de archivo al *file-descriptor* 1.

## n>&m

- `comando 2>&1`

Este operador hace que lo que haya escrito en el *file-descriptor* 2 (*stderr*) vaya al *file-descriptor* 1 (*stdout*)

### El orden importa en bash

Bash evalúa los operadores de redireccionamiento **de izquierda a derecha**.

Por ejemplo si queremos guardar en un archivo el error y salida estandar de un comando se hace asi, por ejemplo para `ls`:

```
ls algo-que-no-existe > archivo 2>&1
```

Gráficamente se puede ver por pasos como funciona esto:

Primero.- Al evaluar bash `> archivo` los *file-descriptors* quedan de esta forma:

```
File descriptor (FD)     Archivo

       0  --------->     /dev/pts/5
       1  --------->     archivo
       2  --------->     /dev/pts/5
```

Hay que notar que `/dev/pts/5` es la terminal y por ende lo esta muestra en la pantalla, se ve que la entrada estandar *stdin* y *stderr* apuntan a la terminal actual `/dev/pts/5`. En cambio *stdout* está apuntando a `archivo`

Luego.- bash evalúa `2>&1` que hara que los *file-descriptors* queden de esta forma:

```
File descriptor (FD)     Archivo

       0  --------->     /dev/pts/5
       1  --------->     archivo
             /
       2  ---´
```

Se ve que `2>&1` hace que lo que esta en *stderr* (file descriptor `2`) se duplique en `1` (se copie en `1`), y antes el file descriptor `1` estaba apuntando a archivo. De esta forma tanto *stderr* y *stdout* se guardaran en archivo.

Se puede ver además que **bash ejecuta primero los operadores de redirección antes que los comandos**.

Otro ejemplo si queremos ver con el comando `less` el error estandar y la

salida estandar de un comando podemos usar lo siguiente:

- `comando1 2>&1 | less`

Esto hace que el *stderr* vaya hacia *stdout* y como `|` redirije *stdout* al *stdin* del comando que esta a la derecha, `less` recibira los dos streams como argumentos.

## Ejemplos de utilidad

En lo que queda de este tutorial se usarán ejemplos concretos para algunos operadores de redirección.

```
# obtener los 10 procesos que usan mas memoria RAM de todos los usuarios
ps aux --sort -rss | head

# obtener los 10 procesos que usan mas memoria RAM del usuario 'lorilu'
ps aux --sort -rss | head | grep -E "^lorilu"
```

En estos dos primeros ejemplos usamos `ps --sort -rss` para que ordene los procesos de todos los usuarios por el campo RSS que se refiere a la memoria residente o la cantidad de memoria RAM que ocupa acutalmente un proceso, usamos `|` para redirigir la *stdout* de `ps` a la *stdin* de `head` que muestra las 10 primeras líneas. Al agregar `| grep -E "^lorilu"` hacemos que `grep` reciba la salida de `head` y filtre solamente las líneas que empiezan en "lorilu" que es el nombre del usuario de quien buscamos los procesos abiertos, se busca solo en el principio de la línea por que `ps` pone el nombre del usuario al principio de cada línea.

```
# itera linea por linea en archivo.txt
while read -r linea
do
   echo "$linea"
   # extas ...
done < archivo.txt
```

Este script muestra que bash ejecuta primero los operadores de redirección (en este caso <) antes que los comandos. En la parte final del script bash, `< archivo.txt` hace que *stdin* apunte al contenido de `archivo.txt` y esto hace que el bucle while itere sobre el contenido del archivo y haga que `read -r` lea cada

linea y guarde el resultado de cada iteración en la variable `linea`, `echo` simplmente muestra la línea actual. Dentro el blucle se pueden poner una cantidad indefinida de comandos para procesar cada línea a conveniencia.

-

```
# renombra los archivos cuyos nombres tienen espacios en blanco y los reemplaza por _
for nombre in *
do
    nombre_sin_espacios=$(echo $nombre | tr " " "_")
    mv "$nombre" "$nombre_sin_espacios" 2> /dev/null
done
```

A veces es problemático tener archivos con nombres con espacios en blanco como "archivo numero 1" lo que hace el script anterior es renombrarlo como "archivo_numero_1".

El bucle `for` actua iterando en `*` que coincide con cualquier archivo dentro el directorio actual y guardando cada nombre de archivo en `nombre` en cada iteración. Luego `echo $nombre | tr " " "_"` hace que `tr` reemplaze cualquier caracter " " por "_" y la salida de esto se guarda en la variable `nombre_sin_espacios`.

Luego se usa `mv "$nombre" "$nombre_sin_espacios"` para renombrar el archivo por su "equivalente" sin espacios en blanco.

La parte `2> /dev/null` es opcional ya que para el caso en que los archivos no tengan nombres con espacios en blanco, `mv` mostrará un error ya que los archivos son del mismo nombre y no pueden renombrarse, ese error va hacia *stderr* y se mostraría en pantalla si no fuese por `2> /dev/null` que pone *stderr* en `/dev/null` haciendo que se pierda y no se muestre en pantalla los errores producidos por `mv`, esto es es como silenciar los errores.

-

```
# Cifra una carpeta con la clave "123456" comprimiendola primero
tar -cf - carpeta/ | gpg --symmetric --no-use-agent --passphrase "123456" >
carpeta.gpg
```

El comando `tar` puede comprimir archivos o carpetas pero requiere escribir el resultado en algun lugar por ejemplo `tar -cf carpeta.tar.gz carpeta/` lo guarda en el archivo `carpeta.tar.gz`. En este caso para enviar su salida directamente a otro programa se usa `tar -` donde `-` es una opción de `tar` para enviar el resultado a *stdout* (salida estándar).

Luego a medida que la carpeta se va comprimiendo, mediante `|` se envía el

resultado a `gpg --symmetric` que cifrará lo que lea de *stdin* y con `> carpeta.gpg` se guarda la salida estándard (*stdout*) en el archivo carpeta.gpg. Las opciones `--no-use-agent --passphrase "123456"` son para indicarle a `gpg` que no le pida al usuario ingresar la clave mediante teclado y en lugar de eso utilice "123456" para cifrar el mensaje.

```
-
# Enviar una carpeta al servidor algo.com a medida que es comprimida
# primero se requiere que en el servidor algo.com se tenga un puerto escuchando
# el trafico en un puerto dado para recibir el contenido enviado

# En el servidor
nc -l -p 5151 | tar -C . -xvf -
```

Primero en el servidor "algo.com" abrimos el puerto 5151 haciendo que `nc` (netcat) escuche el tráfico en este puerto. `nc` enviara a *stdout* el tráfico que reciba satisfactoriamente y con `|` hacemos que `tar -xvf -` descomprima lo que reciba en *stdin* y lo guarde en el directorio actual con las opciones `-C .`

```
# En el cliente (la maquina que envia la carpeta)
tar -cf - carpeta/ | nc -q 0 algo.com 5151
```

Como el servidor esta escuchando el tráfico en el puerto 5151, usamos `tar -cf - carpeta/` para comprimir la carpeta y enviar el resultado a *stdout*. Con `|` lo redirijimos al *stdin* de `nc algo.com 5151` que envía lo que lea del *stdin* al servidor `algo.com` por el puerto `5151`. La opción `-q 0` de netcat hace que cuando se detecte `EOF` (fin de archivo) de la entrada estándard al cabo de 0 segundos se cierre el programa.

De esta forma se puede enviar un archivo mediante `TCP` a medida que es comprimido y al llegar al servidor se ira descomprimiendo, se puede obviamente combinar esto con otros comandos para por ejemplo cifrar el contenido o analizar el envío en tiempo real.

```
-
# convertir una secuencia de imagenes .jpg en un archivo pdf ordenando
alfabeticamente
ls *.jpg | sort -n | tr '\n' ' ' | sed 's/$/\ doc.pdf/' | xargs convert - doc.pdf
```

Solución extraída de [este hilo en stack-overflow]

Con `ls *.jpg` listamos todos los archivos de imagen .jpg, luego `sort -n` los ordena alfabéticamente, la lista ordenada la recibe `tr '\n' ' '` que reemplaza los

saltos de línea `\n` por espacios en blanco , después `sed 's/$/\ mydoc.pdf/'` agrega al final de la línea un espacio en blanco " " y la cadena "doc.pdf" esto por que el comando `convert` requiere recibir las imagenes seguido del nombre del archivo pdf que creará a partir de los archivos de imágenes que le preceden.

Finalmente mediante `| xargs convert - doc.pdf` se hacen varias cosas, primero el comando `xargs` se utiliza para pasarle como **lista de argumentos** la salida *stdout* de lo que esta a la izquierda de `|` al comando que esta a la derecha, en este caso es `convert - doc.pdf` que tomará los nombres de los archivos de imagen y a partir de estos creará el archivo doc.pdf.

Esto es útil cuando se requiren hacer reportes rápidos en pdf de una lista larga de imágenes.

## Lectura adicional

- http://www.catonmat.net/download/bash-redirections-cheat-sheet.pdf (Tabla de referencia rápida sobre operadores de redirección)
- http://wiki.bash-hackers.org/howto/redirection_tutorial (Tutorial extenso sobre redirección en bash)
- https://www.gnu.org/software/bash/manual/html_node/Redirections.html (Manual de referencia sobre redirecciones en bash)