

Historia y trucos de la Consola GNU+Linux

Autor: Arturo 'Buanzo' Busleiman <buanzo@buanzo.com.ar>

Ante todo quiero mencionar el orgullo que representa para mi el escribir un artículo para una revista Española, especialmente SoloLinux, la cual es reconocida en mi país de origen, Argentina. Adicionalmente, deseo dedicar este artículo a mi hermano del alma, Matias, quien llegó a vuestro país en Octubre. Y ahora, los invito a continuar leyendo esta reseña de la historia de GNU+Linux, sus conceptos clave y, por supuesto, comandos. Esta guía no intenta ser la típica guía sobre comandos en GNU+Linux, resumiendo nombre y descripción del comando, sino que trataré de transmitirles conceptos importantes de todo Unix junto con algunos comandos de ejemplo. Por supuesto, agregaré una tabla de comandos y su descripción... pero luego de aclarar todos los conceptos. Que la disfruten!

Un poco de historia...

Corría el año 1965, y en los Laboratorios Bell se utilizaba el sistema operativo Multics, siglas de Multiplexed Information and Computing Service (Servicio Multiplexado de Información y Computación). Decidieron descartarlo en 1969, para reemplazarlo por GECOS. Allí aparecen nuestros héroes Ken Thompson y Dennis Ritchie, que deseaban fervientemente poder jugar al Space Travel en una PDP-7 por DEC. Tuvieron que reescribir un sistema operativo completo para este ordenador tan pequeño. Lo denominaron UNICS, siglas de Uniplexed Information and Computing Service. Claro está, el nombre es un chiste, como lo fue el desarrollo del sistema operativo, tan solo para jugar un juego en una DEC PDP-7. Unix, finalmente, fue desarrollado durante el verano norteamericano de 1969.

Finalmente, en 1971 se libera la primera versión de Unix! Entre sus comandos básicos se encuentran: b, utilizado para compilar programas escritos en dicho Lenguaje B. Entre los clásicos, se podían encontrar a cat, chdir, chmod, chown, cp, ls, mv, wc, who. Es muy interesante destacar un detalle curioso: La mayoría de los comandos son abreviaciones de palabras en idioma Inglés: cp, por CoPy (copiar), mv por MoVe (mover), ls por LiSt (listar), wc por Word Count (contar palabras). ¿Y saben por qué esto es así? Ubíquense en la época! Los sistemas de transmisión de datos eran terriblemente lentos, la respuesta de los sistemas al 'input' de un usuario era aún más lenta. Imagínense tener que tipear comandos con nombres, por ejemplo, como "changemode", "changeowner", o "connectedusers". Si más de una vez nos equivocamos al tipear comandos cortos, imagínense tipear confiadamente, y presionar ENTER al finalizar una secuencia que aún no aparecía en nuestra terminal... para darnos cuenta de que hemos cometido un error de sintaxis! De esta forma, hasta el día de hoy se mantiene el concepto de utilizar acrónimos o abreviaciones para comandos típicamente tipeados en una consola o xterm, como por ejemplo 'joe' por "Joe's Own Editor", un excelente editor de texto para la consola. Una excepción claramente vinculada con el entorno gráfico puede ser KDE: Comandos como 'konqueror', 'korganizer', 'kontakt' y 'konsole' nos dan una idea.

Continuando con nuestra historia, año y medio después de la publicación del GNU Manifesto en 1985 comienza el Proyecto GNU, liderado por Richard Mathew Stallman, a quien probablemente ya conozcan. El objetivo del GNU Project era el de desarrollar un sistema operativo Libre, que respetara las 4 libertades básicas: La libertad 0, de ejecutar un programa, por el propósito que sea. La libertad 1, de estudiar y/o analizar cómo funciona un programa, donde el acceso al código fuente es una condición previa. La libertad 2, de redistribuir copias para ayudar a quienes no puedan obtenerlas y por último la libertad 3, de poder mejorar el programa y

redistribuirlo para el beneficio de la comunidad. Otra vez, el acceso al código fuente es una condición previa.

Este sistema operativo libre estaría basado en Unix. El único problema es que el Proyecto GNU no empezó por un Kernel, núcleo de un sistema operativo, y en 1991 Linus Torvalds presentó un kernel propio, denominado Linux. Los hackers comenzaron a vincular las aplicaciones del proyecto GNU (compiladores, editores de texto, herramientas de administración de archivos, etc) junto con este Kernel Linux, y así comenzó la vida de la pareja más famosa de la historia... al menos de la historia que nos interesa en esta revista!

De esta forma llegamos al sistema operativo GNU+Linux de nuestros amores: Comencemos ahora a conocer mejor su consola, comandos y al shell Bourne Again!

Características de Linux

Linux, como todo Unix que se precie, es Multitarea, Multiusuario y es implementado bajo el concepto primordial de que "Todo es un Archivo". De esta forma, los dispositivos de entrada/salida, almacenamiento y las conexiones de red son tratados como archivos. Linux, a su vez, extiende el concepto presentando a ciertas estructuras y variables internas del Kernel como archivos y directorios. De esta forma se pueden modificar o acceder parámetros de funcionamiento del sistema, tanto en forma manual como programática. Estoy hablando del directorio `/proc`, que utiliza el pseudo-sistema de archivos `procfs`. Una vez que les presente los comandos básicos para navegar la estructura de directorios y leer archivos del sistema podrán descubrir una gran cantidad de información muy útil, tal vez no tanto en sus primeros pasos con el sistema a bajo nivel, pero que si deciden orientarse a la seguridad informática o la programación en Linux les serán de gran utilidad.

Todo sistema operativo multitarea debe incluir herramientas que faciliten la administración y ejecución de los programas a ser ejecutados. De la misma forma, el hecho de que sea multiusuario no solamente indica de que varios usuarios pueden utilizar simultáneamente (o en forma controlada, como el caso de un dispositivo como una impresora) los recursos del sistema, sino de que los programas también funcionarán bajo una identidad concreta: la del usuario que los ejecutó, o la que el administrador (aquí denominado **root** o superusuario) designe.

Alendo un poco más en detalle, en Linux los usuarios obtienen un código de identificación único cuando son creados, es el llamado UID (User ID). Este UID se utiliza todo el tiempo: para establecer privilegios, para controlar el acceso a recursos, para definir la membresía de archivos y directorios, etc. A su vez, un UID está vinculado con uno o más GIDs, que son Group IDs, Identificadores de Grupo. Cuando un usuario es creado, se le otorga un UID y se lo hace pertenecer al menos a un grupo. Los grupos también tienen nombres simbólicos, y son definidos en el archivo `/etc/group`. Los usuarios se encuentran definidos, a su vez, en el archivo `/etc/passwd`, donde se lista su nombre simbólico, UID, GID primario, shell o intérprete de comandos por defecto, entre otros datos.

En `/etc/group` también se especifican los grupos adicionales a los cuales un usuario pertenece, más allá del primario. De esta forma, el primer comando que yo considero que un usuario debe conocer, es el comando **id**, proveniente de la palabra inglesa para "identificación". Ejecútenlo, sin parámetros, en su consola y verán que les entrega la información de qué usuario son y a que grupo[s] pertenecen. En el caso del usuario **root** el UID y el GID equivalen a **cero**. Si desean obtener los

detalles de identificación de otro usuario, tan solo agréguelo como primer parámetro, por ejemplo **id root** o **id noelia**.

La salida de este comando, es similar a la siguiente (tomado de un usuario agregado según la documentación de instalación de Gentoo Linux):

```
buanzo@murray ~ $ id buanzo
uid=1000(buanzo) gid=100(users) groups=100(users),10(wheel),18(audio),35
(games)
```

Aquí podemos ver que el usuario "buanzo" pertenece en forma primaria al grupo cuyo gid es 100, nombre simbólico "users". Su UID es 1000, y adicionalmente pertenece a los grupos wheel, audio y games. El pertenecer al grupo wheel tiene sus privilegios: los que pertenezcan al mismo podrán utilizar el comando **su**, correspondiente a "sustituir usuario", el cual permite al usuario que lo invoca, iniciar sesión como otro usuario, sin abandonar la actual. Generalmente, se lo utiliza para pasar al usuario root, sin tener que cambiar de terminal.

DATO UTIL: Si quieren conocer el formato y/o estructura de un archivo de /etc, directorio donde reside la mayoría de los archivos de configuración, pueden hacer uso del comando **man**. Este comando presenta las páginas del manual del sistema, el cual está dividido en secciones. En el recuadro número uno pueden ver un listado de las secciones comunes. Para ahorrar tiempo, la sección 5 contiene la descripción de las estructuras de los archivos de configuración ubicados en /etc. Por ejemplo, **man 5 passwd** les presentará un informe con la estructura del archivo /etc/passwd. Vean que la sintaxis de man es "man sección página".

Terminales Virtuales y Físicas

Antes de conocer más comandos y conceptos, aprendamos acerca de las terminales. En el principio se utilizaban, y se siguen utilizando, aunque en menor medida, servidores de gran capacidad (mainframes), a los cuales, para utilizarlos, se les conectaban **terminales** que constaban, en su mínima configuración, de un monitor y teclado. No poseían inteligencia propia, ya que eran como teclado y monitor conectado directamente al **mainframe**. Ya que el sistema era multiusuario y multitarea, cada usuario se sentaba en una terminal y realizaba sus tareas. El día de hoy, las PC de escritorio son terriblemente poderosas. Y ocupan muchísimo menos lugar, lo cual era otro motivo para utilizar terminales compactas conectadas a un mainframe.

Como ya deben imaginarse, el proceso de "login" al sistema se realiza siempre ingresando usuario, y luego la clave. El día de hoy varias distribuciones de GNU+Linux permiten definir un "login automático" para cierto usuario. Esto es especialmente útil en entornos gráficos, y cuando el sistema lo utiliza físicamente una sola persona, o se comparte la sesión.

Si vuestro sistema inicia en modo gráfico (X11 u Xorg-X11, con KDE, GNOME u otro entorno), para lograr acceder a la consola deben ingresar la combinación **CTRL, ALT izquierdo y una de las teclas de función**, generalmente limitadas de F1 a F6. El hecho de que podamos elegir cualquier tecla de función en dicho rango esta dado porque por defecto, en el archivo **/etc/inittab**, generalmente se configuran 6 terminales, denominadas VCs (por Terminales Virtuales), o **ttys** (por teletipo). De esta forma, tenemos la tty1, tt2, hasta tty6.

En cambio, si su sistema arranca en modo consola (texto 80x25 o gráfico framebuffer, donde generalmente en la terminal 1 habrá una decoración de terminal particular, como en Gentoo o Novell SuSE Linux), nos encontraremos inicialmente en esta terminal 1, y podremos cambiar entre una y otra con la misma combinación, salve que adicionalmente, podemos evitar el uso de la tecla **CTRL** y también podremos utilizar la combinación **ALT izquierdo + CURSOR. izquierdo o derecho**, si queremos "bajar" o "subir" de terminal, respectivamente.

Login y Procesos

En cada una de esas terminales, ustedes encontrarán un prompt de **Login**: precedido por un mensaje que por defecto es tomado del archivo **/etc/issue**. El programa que presenta el prompt de Login en cada terminal es denominado **getty**. Existen varias diferentes implementaciones de getty, entre las más famosas o utilizadas podemos nombrar a mingetty, mgetty (especial para logins via modem) y agetty.

Una vez que ingresen su nombre de usuario y clave correctos, se les mostrará el contenido del archivo **/etc/motd**, por sus siglas en inglés de Mensaje del Día. Al pie descubrirán el denominado **prompt**, donde podrán ingresar comandos, como **id**, ya explicado. Si ingresan con el usuario **root**, el prompt finalizará con el símbolo numeral (**#**). Si no fuera un numeral, entonces significa que hicieron login con un usuario sin privilegios. Un prompt clásico puede verse de la siguiente manera:

Para el usuario sin privilegios "buanzo":

```
buanzo@murray ~ $
```

Para el superusuario, "root":

```
murray ~ #
```

Como pueden ver, en el primer caso el formato es de **USUARIO@HOSTNAME ~ \$** para buanzo y **HOSTNAME ~ #** para root. El símbolo **~** en Linux representa al directorio **home** de un usuario. El directorio home es aquel dónde el usuario está ubicado al aparecer el prompt luego del proceso de login y donde generalmente tiene permiso de escritura, desde allí y hacia abajo en el árbol de directorios. Para ser más correctos, es uno de los campos definidos en **/etc/passwd**. Todo este comportamiento es configurable, pero toda distribución, en su configuración original, responde a este esquema de funcionamiento.

DATO CURIOSO: Cuando un cracker ataca un sistema, y logra ejecutar comandos arbitrariamente a través de alguna vulnerabilidad, generalmente suele intentar ejecutar y obtener la salida del comando **id**, para saber con que posibles privilegios cuenta, y planear el resto del ataque, hasta lograr obtener acceso como **root**.

Entrada, Salida, Pipes y mezclando todo eso

Tal vez no se dieron cuenta, pero al ingresar la clave, ya se encuentran ejecutando un programa: el shell, encargado de presentarles el prompt, y de recibir instrucciones por parte de ustedes, devolviendo el resultado por pantalla, **o como ustedes lo indiquen**, ya que existe un concepto que pasaré a explicarles:

En Linux existe el concepto de entrada y salida estándar, **stdin** y **stdout** respectivamente. Un programa puede elegir leer datos desde la entrada estándar, o

desde un archivo. De la misma forma, puede elegir escribir sus resultados a la salida estándar, o a otro archivo. Pero existe un tercer tipo de salida: la de errores. Un programa puede tomar un archivo de datos por **stdin**, mostrar el resultado por **stdout**, y una serie de mensajes de error, o de depuración (o incluso de progreso), por **stderr**, la salida de errores. Un claro ejemplo del uso de estos conceptos lo pueden ver en el **shell**, el cual por defecto suele ser /bin/sh, el **Bourne Again Shell**. El bash, como se lo conoce, no es solamente un shell interactivo, sino un lenguaje de programación de scripts. Luego del proceso de login, bash toma los comandos a ejecutar (programas, o comandos internos del lenguaje bash) de **stdin**, que en este caso suele ser el teclado. Mostrará tanto **stdout** como **stderr** en la pantalla, simultáneamente.

Por otra parte, cuando se programa un script de bash, se lo almacena en un archivo de texto (usualmente con la extensión **.sh**, aunque no es obligatorio). En la primera línea del script se coloca el texto **#!/bin/sh**. Linux reconoce esa secuencia como "Ejecutar /bin/sh y reemplazar el stdin por el contenido del archivo". Dicho de otra forma, la primer línea define el **intérprete** del script, ya que existen numerosos shells y lenguajes de scripting diferentes: tcsh, pdksh, python, perl, php, ruby, etc.

Para comprender **stdout** con un ejemplo, realicemos el siguiente ejercicio:

Pueden utilizar el comando **ps -ax** (por Process Show, recuerden usar el comando man!) para obtener un listado de los procesos . Por definición, un proceso es la "copia de un programa puntual, iniciado bajo una identidad y haciendo uso de recursos". Cada proceso tiene su propio "contexto" (la identidad, o sea, usuario y grupos vinculados, archivos abiertos, memoria virtual asignada, etc), pero ante todo, cada proceso posee un identificador que le es propio, denominado **PID**. Con el PID de un proceso se pueden realizar dos tareas, entre muchas, interesantes: Obtener información sobre el mismo, mediante el directorio **/proc**, y enviarle una **señal**, mediante el comando **kill**.

Como habrán podido observar, la "salida" del comando **ps -ax** es mas bien extensa, supera la cantidad de líneas que se muestran simultáneamente en la pantalla. Por este motivo, vamos a ejecutar ps -ax **redirigiendo su stdout** a un archivo de texto. Dicha redirección la realizaremos dejando un espacio luego del último parámetro, luego ingresando el símbolo > seguido del archivo a crear. Por ejemplo:

```
buanzo@murray ~ $ ps -ax > /tmp/salida
```

De esta forma, hemos creado en el directorio /tmp, dónde todos los usuarios pueden escribir, un archivo llamado "salida", creado en base a lo que ps -ax generalmente muestra por la pantalla. Ahora faltaría que lo pudieramos ver, no? A tal efecto vamos a presentar el comando **cat**:

```
buanzo@murray ~ $ cat /tmp/salida
```

Por supuesto, el contenido de /tmp/salida sigue yéndose fuera de la pantalla. Necesitamos algo más poderoso. Utilizaremos el comando **more** (pasen de página con la barra espaciadora, pueden salir tecleando **q**):

```
buanzo@murray ~ $ more /tmp/salida
```

Pero es un programa tosco, no como el de **man**, que nos permite movernos con los cursores, las teclas de paginado Re. Pág y Av. Pag, y hacer búsquedas con **/...** Les presento al programa **less** (salen con **q**):

```
buanzo@murray ~ $ less /tmp/salida
```

Pero, ¿tantas molestias tan solo para poder leer tranquilos la salida de **ps -ax**? ¡Podríamos haber reemplazado el **stdin** de **less** por el **stdout** de **ps -ax**! - La conexión de **stdout** de un programa con el **stdin** de otro se denomina **piping**, traducción de entubamiento. Lo realizamos de la siguiente manera:

```
buanzo@murray ~ $ ps -ax | less
```

De esta forma, nos ahorramos el crear un archivo, y luego tener que leerlo. Ni hablar de eliminar el archivo después de utilizarlo (lo cual haríamos con el comando **rm**, siglas de remove).

```
buanzo@murray ~ $ rm /tmp/salida
```

Pero también existe la posibilidad de que ustedes deseen **agregar** información a un archivo ya existente. Cuando utilizan el símbolo **>** de redireccionamiento, el archivo al cual estén redirigiendo ya existiera, sus contenidos se perderían, serían reemplazados por la salida del comando en cuestión. En cambio, si utilizan el mismo símbolo por duplicado, **>>**, la salida se agregará al final del archivo. Por ejemplo:

```
buanzo@murray ~ $ date > ~/.plan
```

```
buanzo@murray ~ $ echo "Nota Mental, estudiar python" >> ~/.plan
```

En este ejemplo, utilizamos **date** para obtener la fecha y hora actuales, y redirigimos dicha salida al archivo **.plan** en nuestro directorio home (gracias al símbolo **~**). Luego, agregamos la frase **Nota Mental, estudiar python** a ese mismo archivo.

Ahora, si quisieramos escribir algo de varias líneas a un archivo, podríamos hacer uso de dos tipos de redireccionamiento simultáneamente. Vean este ejemplo:

```
buanzo@murray ~ $ cat << FINAL > /tmp/texto
```

Esto es un texto siendo ingresado al **stdin** de "cat" a través del teclado. Cuando escriba la palabra **FINAL** al comienzo de una línea nueva cat lo mostrará a través de su **stdout**, que en este caso está redirigido a **/tmp/texto**. Allí estará almacenado todo este mensaje. Y ahora, a terminar.

```
FINAL
```

```
buanzo@murray ~ $
```

Como pueden ver, al escribir **FINAL** en una línea nueva, y apretar **ENTER**, volvemos al prompt. Ahora el archivo **/tmp/texto** existe, y contiene el mensaje que hemos ingresado:

```
buanzo@murray ~ $ cat /tmp/texto
```

Esto es un texto siendo ingresado al **stdin** de "cat" a través del teclado. Cuando escriba la palabra **FINAL** al comienzo de una línea nueva cat lo mostrará a través de su **stdout**, que en este caso está redirigido a **/tmp/texto**. Allí estará almacenado todo este mensaje. Y ahora, a terminar.

```
buanzo@murray ~ $
```

Por supuesto, sin la palabra "FINAL". Ustedes van a ver estos ejemplos en Internet con la palabra **EOF**, que significa Fin de Fichero. El detalle es que no importa que palabra de cierre utilicen luego del símbolo **<<**, solo que deberán recordar no utilizarla en una línea nueva, de a no ser que quieran finalizar. Esta secuencia es muy útil en scripts de bash, donde tal vez necesitemos crear un archivo de texto con

una cierta estructura.

Continuando con los ejemplos de redireccionamiento, en vez de decirle a cat que archivo leer, pasémoslo por stdin:

```
buanzo@murray ~ $ cat < /tmp/texto
```

Esto es un texto siendo ingresado al stdin de "cat" a través del teclado. Cuando escriba la palabra FINAL al comienzo de una línea nueva cat lo mostrará a través de su stdout, que en este caso está redirigido a /tmp/texto. Allí estará almacenado todo este mensaje. Y ahora, a terminar.

```
buanzo@murray ~ $
```

Es muy común que un comando nos muestre errores en pantalla. Es altamente probable que querramos eliminarlos, si es que son errores que no nos afectan. Por ejemplo, el comando find cuando lo utiliza un usuario sin privilegios para buscar un archivo desde la raíz del sistema, /, se encontrará con que no tiene permiso para acceder a ciertos directorios. Find presenta en este caso un mensaje de error. Para eliminar dichos mensajes, que pueden ocupar varias pantallas, podemos utilizar la siguiente redirección:

```
buanzo@murray ~ $ find / -name algun.archivo.txt 2>/dev/null
```

De esta forma hemos redirigido la salida de errores, mediante el uso del operador **2>**, a la entrada del pseudo-dispositivo **null**. Dicho "archivo" puede ser leído, también, pero no obtendremos ninguna salida del mismo.

Por ejemplo, podríamos crear un archivo vacío, de cero bytes, de la siguiente forma:

```
buanzo@murray ~ $ cat /dev/null > archivo_a_crear
```

Continuando con los errores de find, recuerden que los mismos generalmente se escriben a stderr, que por defecto NUNCA se conecta con el stdin de otro programa, al utilizar el pipe. Existe la posibilidad de redirigir stderr a stdout, sumando así al stdout normal, o podemos también "apagar" el stdout, y mostrar stderr a través del stdout.

Para sumar stderr al stdout:

```
buanzo@murray ~ $ find / -name algun.archivo.txt 2>&1
```

Para eliminar stdout, reemplazándolo por stderr solamente:

```
buanzo@murray ~ $ find / -name algun.archivo.txt 2>&1 >/dev/null
```

Claro está, si quisieramos redirigir la salida a un archivo, reemplazaríamos el 2>&1 por 2>/tmp/algun.archivo. Si quisieramos reenviar toda esta salida stdout hacia el stdin de otro programa, tan solo deberíamos utilizar el pipe, por ejemplo, con less, para no perdernos los errores:

```
buanzo@murray ~ $ find / -name algun.archivo.txt 2>&1 >/dev/null | less
```

Volviendo a los pipes, uno de los comandos que comúnmente más van a encontrarse utilizando a la derecha del pipe, es el comando **grep**. El mismo nos provee la función de leer stdin, o el archivo que le indiquemos, en busca de uno o más patrones, mostrándonos en stdout las líneas que concuerden. Este comportamiento se puede invertir, para que nos muestre las líneas que NO concuerden.

Para avanzar un poco esta guía, voy a presentarles varios comandos utilizados

simultáneamente, a efectos de obtener de la salida de **ps ax**, el PID de todos los procesos llamados ssh, evitando el demonio sshd.

```
buanzo@murray ~ $ ps ax | grep ssh | grep -v sshd | grep -v grep | cut -c-5 | xargs  
8230 8350
```

ATENCIÓN: Mejoraremos este comando luego de analizar cada uno de sus elementos.

Un dato curioso: El último comando en ejecutarse en el ejemplo anterior, es... ps ax. El primero es xargs. Esto ocurre porque como cada comando se queda esperando que le llegue un EOF por stdin. Por lo tanto, si en vez de poner ps ax como primer comando, pusieran tan solo cat, a secas, y pasaran a otra terminal, en la salida de ps ax verían al xargs, cut -c-5, los tres grep y al cat funcionando simultáneamente.

Veamos cada comando de la secuencia en detalle. Como ejercicio, vayan ejecutando lo siguiente a medida que lo vamos analizando. Si con ssh no obtienen resultados, tal vez sea porque no hay ninguna conexión ssh en progreso. Aprovechen, y adapten el comando a sus necesidades!

Ps ax, ya lo conocemos, nos muestra todos los procesos:
buanzo@murray ~ \$ ps ax

En este caso, de la salida de ps ax, deseamos que grep solo tome las líneas que contengan la cadena "ssh":
buanzo@murray ~ \$ ps ax | grep ssh

Se da la casualidad de que ese listado también incluiría a todo programa que contenga la cadena "ssh", como por ejemplo, al "sshd" que es el servicio de ssh, y no el programa cliente. A tal efecto, sacaremos todas las líneas que contengan sshd, por lo que utilizaremos el parámetro "-v" de grep, para inVertir la salida:
buanzo@murray ~ \$ ps ax | grep ssh | grep -v sshd

El único problema... es que también se incluiría al "grep ssh", ya que contiene la cadena "ssh"! Por lo tanto, debemos eliminar las líneas que contengan "grep", volviéndonos a utilizar "-v":
buanzo@murray ~ \$ ps ax | grep ssh | grep -v sshd | grep -v grep

Bien, ya logramos separar las líneas de **ps ax** que nos interesan. Ahora solo debemos extraer el valor de PID de cada línea, que corresponde a los 5 primeros caracteres de cada línea. El comando **cut** nos permite tomar porciones de una línea, o campos en base a un separador específico (veremos un ejemplo más adelante). En este caso, el parámetro "-c" nos permite especificar el rango de caracteres a tomar. En este caso, usaríamos "-c1-5", para tomar los caracteres del 1 al 5 inclusive. Ya que empezamos por el uno, podemos omitirlo, y tan solo formarlo por "-c-5".
buanzo@murray ~ \$ ps ax | grep ssh | grep -v sshd | grep -v grep | cut -c-5

Claro, en última instancia nos interesaría tal vez ver el resultado en una sola línea, en vez de un PID por línea. Xargs, al ejecutarse sin ningún parámetro, realiza dicha función:
buanzo@murray ~ \$ ps ax | grep ssh | grep -v sshd | grep -v grep | xargs

Pero xargs también nos permite ejecutar un comando con cada uno de esos valores de PID! Por ejemplo, para matar un proceso, solo necesitamos conocer su PID, y hacer uso del comando **kill**. Existen dos formas de matar un proceso: solicitándole

que finalice, o verdaderamente matándolo. Eso depende de todo un tema muy interesante denominado "Administración de Señales", que se escapa al alcance de ese artículo. En nuestro caso, usaremos kill a secas, lo cual "solicita" a un cierto PID que finalice, sin forzar su muerte. Agregaremos el parámetro "-n 1" al xargs, para indicarle que tome UN pid por vez, y ejecute por cada uno el kill. Por ejemplo:

```
buanzo@murray ~ $ ps ax | grep ssh | grep -v sshd | grep -v grep | xargs -n 1 kill
```

De todas formas, kill permite ingresar múltiples PID como parámetros, por lo que el "-n 1" no es verdaderamente necesario.

Piensen lo siguiente... si cada vez que quisieramos matar un grupo de procesos llamados de cierta forma concreta tuvieramos que realizar toda esta secuencia, no solo podríamos llegar a cometer un error, incluyendo un proceso no deseado, sino que sería una molestia. A tal efecto, se desarrolló el comando "killall", que en vez de tomar un PID, toma el nombre de un programa. Por lo tanto "killall ssh" hubiera sido más que suficiente.

El comando "grep -v sshd | grep -v grep" puede compactarse en un sólo comando, ya que grep permite utilizar expresiones booleanas en los parámetros de búsqueda, de la siguiente forma: **grep -v -E 'sshd|grep'**. En este caso, el símbolo de pipe está siendo incluido dentro de comillas simples, las cuales ocultan el significado especial de todos los símbolos en la categoría "metacaracter". Entre estos símbolos, que son interpretados por bash, encontramos a asterisco, el signo de pregunta, el signo de admiración y el símbolo ampersand, entre otros. El significado especial ahora debe ser analizado por grep, en este caso, corresponde a la operación booleana "OR", que resulta positiva si uno, otro o ambos elementos se encontraran. La construcción, por lo tanto, significa "eliminar del stdout toda línea que contenga sshd o grep, leída del stdin".

Algunos ejemplos avanzados con find

El comando find permite realizar búsquedas en el sistema de archivos, desde el directorio que le especifiquemos. Entre los parámetros de búsqueda podemos especificar: la cantidad de subdirectorios a profundizar en la búsqueda, expresiones regulares en el nombre de archivo, teniendo en cuenta o no mayúsculas y minúsculas, tipo de archivo (directorio, link simbólico, archivo, dispositivo, pipe, etc). Incluso podemos ejecutar un programa por cada archivo que se encuentre (o alimentar el stdin de xargs con la salida de find, también!).

Un ejemplo muy útil, al menos para mi, es cuando quiero revisar mi colección de videos digitalizados, es ingresar en modo gráfico, abrir una terminal, y ejecutar mi aplicación favorita para visualizar películas, mplayer. En mi caso particular uso find con el parámetro -exec para ejecutar una aplicación por cada archivo encontrado, pasándole al mplayer el path completo a ese archivo. Por ejemplo:

```
buanzo@murray ~ $ find /downloads -iname "*.mpg" -exec mplayer -fs -- {} \;
```

Find buscará desde /downloads todo archivo cuya extensión sea mpg, sin distinguir entre mayúsculas y minúsculas gracias al parámetro -iname, y ejecutará el comando mplayer -fs, reemplazando las doble llaves por el path completo al archivo encontrado. El parámetro -exec finaliza con "\;"

Sumando un poco de scripting al find

Como ven, estamos limitando las extensiones a ".mpg". Con un simple bucle for, en el cual podremos ejecutar el comando find cambiándole el parámetro -iname por cada ciclo del bucle, de esta forma: Crearemos la variable f, leída como \$f, y la utilizaremos dentro del parámetro -iname.

```
buanzo@murray ~ $ for f in mpg mpeg avi asf wmv; do find /downloads -iname "$f"
-exec mplayer -fs -- {} \; ; done
```

De esta forma, \$f obtendrá el valor mpg, mpeg, avi, asf y wmv. Equivaldría a la ejecución de cinco "finds" diferentes.

Algunos detalles de Bash

Bash guarda un registro de todos los comandos que hemos ejecutado, hasta un cierto límite. Pueden obtener un listado de estos comandos con "history". Cada comando tiene un número asignado. Pueden volver a ejecutar un cierto comando tipeando el signo de admiración seguido del número correspondiente y presionando ENTER, por ejemplo, "!78".

Si estan ejecutando un programa que lee stdin, pueden salir del mismo presionando CTRL+D, que representa al código EOF, Fin de Fichero.

Si agregan el símbolo & al final de un comando, el comando se ejecutará "de fondo", y obtendrán prompt inmediatamente. Si ejecutan un programa y quieren llevarlo al fondo, presionen CTRL+Z. El comando **jobs** les mostrará un listado de todos los comandos funcionando en background, y con el comando **fg**, seguido del número de job, pueden "traerlo al frente", y trabajar con el mismo. Para cancelar un programa, utilicen CTRL+C.

Si desean ejecutar un comando dependiendo de si otro ha finalizado exitosamente, o no, pueden utilizar el doble ampersand y el doble pipe.

El comando2 se ejecuta si comando1 es exitoso:
buanzo@murray ~ \$ comando1 && comando2

El comando2 se ejecuta si comand1 NO es exitoso:
buanzo@murray ~ \$ comando1 || comando2

La página del manual de bash tiene 4789 líneas. Este artículo tiene aproximadamente 600. Bash es simple, poderoso y lleno de funcionalidad.

Finalizando

Hemos conocido como funciona el stdin, stdout y stderr. Como se vinculan y redirigen varios programas entre si, como realizar búsquedas y ejecuciones con find, pero esto es sólo la punta del iceberg: deben, con estos conceptos, realizar sus propias pruebas, sus propias combinaciones y sus propias investigaciones. Revisar la página del manual de cada comando es lo primero que deben hacer.

Para finalizar, entonces, les dejo una tabla de los comandos más utilizados y una descripción de la función que cumplen, en el recuadro número dos. Podría mostrarles muchos comandos, pero la filosofía de Unix es la de tener herramientas

chicas y concretas que, combinadas, produzcan resultados y solucionen problemas. Con esta introducción y con estos conceptos claros, no van a tener problemas en comprender y utilizar los siguientes comandos.

Los invito a que nos cuenten sus experiencias al investigar la consola. Han quedado temas en el tintero, pero estoy convencido de que no faltará oportunidad para tocar dichos temas en un próximo artículo. Los invito a todos a visitar mi sitio personal, www.buanzo.com.ar, dónde encontrarán recursos adicionales.

RECUADRO NUMERO UNO - "Secciones del Manual"

1. Programas o comandos de shell
2. Llamadas al Kernel
3. Llamadas a Librerías
4. Archivos especiales
5. Convenciones y formato de archivos
6. Juegos
7. Macros
8. Administracion del Sistema
9. Rutinas del Kernel
- n. TCL/TK

RECUADRO NUMERO DOS - "Comandos Comunes y su Descripción"

basename	Extrae el nombre de archivo de un path.
bzip2	Comprime y descomprime archivos. Excelente compresión.
cd	Cambia de directorio. No olviden de separar el directorio con un espacio.
cal	Muestra un calendario del mes actual, o de cierto mes/año
chmod	Cambia los permisos de lectura, escritura, ejecución y especiales de archivos o directorios.
cp	Copia archivos y directorios a un directorio/archivo destino
dd	Permite leer y escribir a bajo nivel un archivo o dispositivo. Util para armar imagenes ISO de particiones, o de CD-ROMs, entre otros usos.
df	Nos muestra la cantidad de espacio libre en cada partición montada.
du	Nos muestra cuanto espacio ha sido utilizado en cada partición montada.
export	Permite definir variables de entorno para el shell actual y sub-shells.
file	Mediante el uso del archivo /usr/share/misc/file/magic.mgc identifica un tipo de archivo.
groupadd	Agrega un grupo a /etc/group
groupdel	Elimina un grupo de /etc/group
groupmod	Modifica un grupo existente de /etc/group
groups	Lista los grupos a los cuales pertenece un usuario
gzip	GNU zip, comprime y descomprime archivos.
head	Permite leer una cierta cantidad de líneas del principio de un archivo.
halt	Inicia el proceso de apagado del sistema. Equivalente a shutdown -h
host	Permite realizar consultas DNS.
ln	Construye links entre archivos. Un hardlink es un nombre adicional para un archivo. Un softlink apunta a otro archivo. Si se borra el destino, el link sigue existiendo, pero esta "roto".
locate	Permite ubicar un archivo en todo el sistema de archivos, mediante una base de datos armada con updatedb
which	Busca un programa en el PATH e indica todas las ocurrencias.
mount	Permite montar y desmontar particiones, CD-ROMS, USB sticks, etc.
mv	Permite mover archivos y directorios a otras ubicaciones. También utilizado para renombrar (ver man rename).
mesg	Permite habilitar/deshabilitar la recepción de mensajes via el comando write.
passwd	Utilizado para cambiar la clave de acceso de uno mismo, o de otros usuarios si lo utiliza root.
rm	Elimina archivos y directorios
read	Permite leer texto y almacenarlo en una variable de entorno.
rmdir	Elimina directorios vacios.
rpm	Las distribuciones como SuSE y Fedora utilizan RPM para administrar la instalación y actualización de paquetes. Este es el comando principal.
sort	Lee stdin o un archivo, y lo escribe ordenado a stdout.
sleep	Espera una cierta cantidad de segundos.
tar	Permite almacenar una estructura de archivos y directorios en un archivo, opcionalmente comprimiendo con gzip (-z) o bzip2 (-j).
tail	Permite ver archivos de texto a medida que se les agrega información, como un log del sistema (/var/log/*), o solo las últimas N líneas deseadas.
uniq	Elimina líneas duplicadas de un archivo previamente procesado por el comando sort.
uname	Presenta información del sistema: versión de kernel, arquitectura.
uptime	Presenta el tiempo que el sistema lleva encendido.

umount Inverso de mount.
useradd Agrega usuarios al sistema, archivo /etc/passwd
userdel Elimina usuarios del sistema, archivo /etc/passwd
usermod Modifica usuarios del sistema, archivo /etc/passwd
vi Junto con emacs, ed, joe, pico, nano y jed, uno de los editores más
utilizados en la consola de GNU+Linux
whereis Similar a which y locate.
who Junto con w, muestra un listado de los usuarios conectados al sistema.
write Envía mensajes de texto a otros usuarios conectados al sistema. Wall
permite enviar mensajes a todos los mensajes.
whoami Nos indica nuestro nombre de usuario solamente.
zcat cat de archivos comprimidos con gzip, bzip2 u compress. Primero los
descomprime y luego los muestra en pantalla.