GUION 4
MISCELÁNEA

Bibliografía

Sarwar, S. M.; Koretsky, R.; Sarwar, S. A. El libro de LINUX. Addison Wesley, 2005.
Sobell, Mark G. Manual práctico de Linux. Comandos, editores y programación Shell. Anaya Multimedia, 2008
Sánchez Prieto, S. UNIX y LINUX. Guía práctica (Tercera edición). Ra-Ma, D.L. 2004.

1. ÓRDENES RELACIONADAS CON PROCESOS

Linux es un sistema **multiproceso,** es decir, el sistema es capaz de gestionar más de un proceso simultáneamente. En el momento de conectarnos se crea un proceso asociado al *shell* (intérprete de comandos), cuya primera acción es visualizarnos el prompt del sistema y ponerse a la espera de las instrucciones que vayamos a darle a través de nuestro terminal (E/S estándar del proceso). Esto quiere decir que siempre hay un proceso asociado a cada conexión, como mínimo, en curso de ejecución y que sólo desaparecerá mediante la desconexión. Toda orden que le transmitamos se ejecutará mediante un subproceso específico, diferente del shell de partida. El shell de conexión se bloquea a la espera de la señal de fin del subproceso para poder visualizar de nuevo el prompt y ponerse otra vez a la espera de la siguiente orden.

Cada proceso siempre se genera por otro proceso superior (proceso padre). Cada uno de ellos recibe un identificador del sistema; es el PID (*process identification*), que es un número entero único que nos permite localizarlo en el listado de los procesos.

El mandato **ps** ofrece información sobre los procesos iniciados por el usuario desde el terminal.

Sintaxis: ps [-l]

La información dada es variada y depende del "shell" que utilicemos, aunque básicamente ofrece el identificador del proceso (PID), el terminal desde el cual se inició el proceso (TTY), la cantidad de tiempo que el proceso lleva ejecutándose (TIME) y la línea de orden que se escribió para iniciar el proceso (COMMAND). Si se emplea la opción **-l** proporciona más información como, el estado del proceso (S) que puede ser activo o ejecutando (**O**), bloqueado (**S**) y listo (**R**); el identificador del usuario que ha inicializado el proceso (UID, *User Identification*), el identificador del proceso superior (PPID) y la prioridad de ejecución del proceso (PRI) (Esta prioridad está asignada por el sistema operativo.).

```
Ejemplos:
        ps
        ps -l
```

Hay dos formas de ejecutar un trabajo:

1. De forma **interactiva** (*foreground* o en primer plano), que es la que venimos utilizando. Funciona casi como un diálogo de preguntas/respuestas: le damos una orden al shell y esperamos a que nos devuelva el resultado antes de darle otra orden.

2. Con **prioridad subordinada** también denominada tarea de fondo (*background* o en segundo plano), que significa lanzar la orden al shell y continuar nuestro trabajo sin esperar a que nos dé el resultado de la misma. Esta forma de ejecutar un trabajo es bastante útil cuando dicho trabajo requiere bastante tiempo y poca interacción con el usuario. Además, los procesos en segundo plano se ejecutan con menor prioridad que los procesos en primer plano.

Para ejecutar un trabajo (mandato) en tarea de fondo, sólo hay que concluir dicho mandato con el carácter especial **&**. Cuando se solicita que se ejecute un mandato de esta forma, inmediatamente se presentan dos números: uno entre corchetes que indica el número de tarea que ocupa ese trabajo entre los que se están ejecutando como tarea de fondo, y un segundo número que corresponde al identificador (PID) del proceso que ejecuta dicho trabajo. Este identificador interesa tenerlo en cuenta, porque es el número que permite conocer después cómo se desarrolla el proceso en plena actividad. Se puede observar que el shell visualiza a renglón seguido el prompt de espera de nueva orden.

Por consiguiente, es posible ejecutar varios mandatos sucesivos en tarea de fondo, sin dejar el modo interactivo del shell. Pero no es recomendable excederse, puesto que se debilitaría la eficiencia del sistema y se perdería las ventajas de ejecutar de este modo.

```
Ejemplo:
        man more ls cat grep finger find cut paste >ayudas &
```

Sólo el trabajo en primer plano puede utilizar la entrada del teclado. Para conectar el teclado a una tarea que se está ejecutando en segundo plano, es necesario usar el comando **fg** o un signo de porcentaje (%).

Sintaxis: fg [[%]número de la tarea | comando usado para lanzar el trabajo]

Si se omite el número de la tarea o el comando empleado para lanzar el trabajo, la orden **fg** conecta el teclado con el último trabajo lanzado como tarea de fondo. El signo de porcentaje (%) no emplea argumentos y siempre conecta el teclado con el último trabajo subordinado.

El shell mostrará el comando utilizado para comenzar la tarea y se podrá introducir cualquier entrada que el programa necesite para continuar.

Cuando se ejecuta un programa en primer plano, es posible, que se necesite suspender para volver al

shell, hacer algo en él y volver después al proceso suspendido. Por ejemplo, se está editando un programa escrito en C, con el editor *vi*, y se necesita compilar dicho programa para comprobar si se han corregido ciertos errores. Entonces, se pueden guardar los cambios efectuados en el programa, suspender el editor *vi*, compilar el programa para ver los resultados de la compilación y volver de nuevo al editor. Para ello, se suspenderá el proceso de primer plano pulsando *Control*+z, y para poner de nuevo en primer plano un proceso suspendido se empleará el comando **fg**. Por lo tanto, se puede suspender el editor vi pulsando *Control*+z, compilar el programa para identificar los posibles errores y reanudar la sesión suspendida del editor empleando la orden **fg**.

Ejemplo:
Siguiendo estos pasos se puede suspender el editor vi, que es un proceso en primer plano, y posteriormente reanudarlo.

1. Usando el editor vi se escribe el siguiente programa almacenándolo en el fichero calculadora.c

```
$vi  calculadora.c
        #include <stdio.h>
        #include <stdlib.h>

        int main ()
        {
                int num1, num2;

                printf ("Introduce primer numero: ");
                scanf ("%d", &num1);
                printf ("Introduce segundo numero: ");
                scanf ("%d", &num2);

                return 0;
        }
```

2. Se guarda el contenido del fichero.
3. Se suspende la ejecución del editor vi pulsando *Control+z*
4. Se compila el programa usando el compilador gcc: *$gcc calculadora.c*
5. Se ejecuta el programa: *$./a.out*
6. Se reanuda el editor vi pasándolo a primer plano con el comando *fg*
7. Se completa el programa introduciendo el siguiente código:

```
printf ("\nSuma: %d\n", num1+num2 );
printf ("\nRecta: %d\n", num1-num2 );
printf ("\nProducto: %d\n", num1*num2 );
```

8. Se repiten los pasos 2, 3, 4, 5 y 6.
9. Se finaliza el editor vi.

El comando **jobs** permite conocer el estado de los procesos suspendidos y en segundo plano.

Sintaxis: jobs [-l] [[%]lista de números de tarea]

Si no se especifica la lista de números de tarea, el comando **jobs** muestra el estado de todas las tareas en curso. Con la opción l, también se muestra el PID de los procesos asociados a esas tareas.

Ejemplo:
En el ejemplo anterior, se puede ejecutar el comando jobs una vez suspendido la ejecución del editor vi, es decir, después del paso 3 y antes del paso 6.

El comando **kill** permite terminar un proceso en curso. Para ello es preciso saber el identificador que le corresponde. Este mandato envía la señal de finalización al proceso.

Sintaxis: kill PID

```
Ejemplo:
Suponiendo que el PID del proceso asociado a la tarea de fondo lanzada con el siguiente
comando:
       man more ls cat grep finger find cut paste >ayudas &
es 1185, este proceso se podría eliminar, si todavía no se ha completado, con el comando:
       kill  1185
```

2. LOCALIZACIÓN DE COMANDOS

Cuando introducimos un comando en Linux, el shell busca en una lista de directorios el programa cuyo nombre coincide con el comando introducido y ejecuta el primero que encuentra. La lista de directorios se llama ruta de búsqueda. Si no se cambia esta ruta, el shell buscará sólo en un conjunto determinado de directorios y después se detendrá. El resto de directorios del sistema también pueden contener utilidades.

which. Muestra la ruta absoluta de una utilidad o comando

Esta utilidad permite localizar la ubicación completa de una utilidad o comando dentro de los directorios de la ruta de búsqueda. Si en la ruta de búsqueda se encuentra más de un programa con el nombre del comando especificado, **which** sólo muestra la ruta completa del primero, ya que será el que se ejecuta.

Sintaxis: which nombre_comando

```
Ejemplos:
       which  vi
       which  ps nice
```

whereis. Muestra los archivos relacionados con una utilidad o comando

Esta utilidad busca en la lista de directorios estándar los archivos relacionados con una utilidad.

Sintaxis: whereis nombre_comando

```
Ejemplos:
       whereis  find
       whereis  bzip2  mesg
```

find. Localiza un archivo dentro de una estructura de directorios

Este comando examina toda la estructura de directorios que se especifique, buscando los archivos que cumplan los criterios señalados en la línea de órdenes. Una vez localizados, podemos hacer que ejecute distintas acciones sobre ellos.

Sintaxis: find directorio(s)_busqueda -[name|mtime|size|type] expresión

directorio(s)_busqueda: directorios del sistema de archivos. El argumento *expresión* se usa para indicar los criterios de selección de los archivos a localizar, pudiendo usarse los caracteres comodín (*, ?).

Opciones:

1. name: únicamente se buscan los archivos cuyo nombre se especifica con *expresión*;
2. mtime n: se buscan los archivos que han sido modificados hace *n* días;
3. mtime –n: se buscan los archivos modificados en los últimos *n* días;
4. mtime +n: se buscan los archivos modificados hace más de *n* días;
5. size n: se buscan los archivos que usan exactamente n bloques de datos (tamaño predeterminado de 512 bytes, pero se puede especificar la unidad añadiendo w, k, M o G);
6. size +n: se buscan los archivos que usan más de n bloques de datos;
7. size -n: se buscan los archivos que usan menos de n bloques de datos;
8. type x: se buscan los archivos del tipo x, donde x puede ser: f archivo regular, d directorio, …

```
Ejemplos:
       find -name prueba
       find ~ -name prueba
       find / -name usuarios
       find /usr /lib -name "sol*"
       find /usr -size 10k
       find /usr/share  -type d -size -12k
       find -mtime 7
```

script. Guarda en un fichero toda la actividad de un terminal

Esta utilidad permite guardar en un fichero todo lo que se muestra en pantalla, así como los inputs y los outputs, hasta que se ejecute el comando **exit**.

Sintaxis: script [nombre_fichero]
script [-a] nombre_fichero

Opciones:

- *nombre_fichero*: fichero en el que se guardarán los datos del terminal. En caso de no especificar ninguno, se guardarán por defecto en un fichero denominado *typescript*.
- La opción **–a** sirve para añadir la información al final de un fichero ya existente.

```
Ejemplos:
       script log.txt
       script –a log.txt
```

3. COMPRIMIR Y DESCOMPRIMIR ARCHIVOS

Linux ofrece algunas utilidades para comprimir/descomprimir ficheros.

bzip2. Comprime un archivo

Esta utilidad permite comprimir un archivo (que no sea un enlace o directorio) analizándolo y

almacenándolo de una forma más eficaz. La nueva versión del archivo tendrá la extensión *bz2* recordando que el archivo está comprimido y que no se puede visualizar hasta que no se descomprima.

Sintaxis: bzip2 [-vkd] nombre_fichero (o lista de nombres de ficheros)

Opciones:

- **v**: informa del porcentaje que puede reducir el tamaño del archivo.
- **k**: crea un nuevo fichero comprimido sin sobrescribir el fichero original, dándole a este nuevo fichero el mismo nombre que el fichero original con la extensión .bz2.
- **d**: descomprime el fichero, realizando la misma función que el comando bunzip2.

```
Ejemplos:
       bzip2 –v datos
       bzip2 -d datos.bz2
       bzip2 -k datos
       bzip2 -vk misdatos1 misdatos2
```

bunzip2. Descomprime un archivo

Esta utilidad restaura el archivo que se ha comprimido con bzip2.

Sintaxis: bunzip2 fichero_comprimido.bz2 (o lista de ficheros comprimidos)

```
Ejemplos:
       bunzip2 datos.bz2
       bunzip2 misdatos1.bz1 misdatos2.bz2
```

4. EMPAQUETAR Y DESEMPAQUETAR ARCHIVOS

La utilidad **tar** tiene dos funciones:

1. almacenar (empaquetar) en un único archivo (llamado archivo tar) todos los ficheros y directorios (junto con su jerarquía de ficheros y directorios) especificados.

   Sintaxis: tar -[vcr]f nombre_fichero.tar nombre_fichero_directorio
   (o lista de nombres de ficheros y directorios)

   Opciones:

   - **f** (write): este modificador debe situarse siempre al final de la lista de modificadores e indica que se utilice el nombre_fichero.tar como archivo para empaquetar. Si no se emplea esta opción, el valor predeterminado para archivar es /dev/mto (cinta magnética), dispositivo que puede no estar configurado en la máquina o al que no siempre se tiene permiso para su uso.
   - **v** (verboso): sirve para que se visualicen los archivos y directorios que se van a archivar;
   - **c** (create): crea un nuevo fichero tar donde se empaqueta la estructura especificada;
   - **r** (append): añade los archivos al final del archivo tar actual.

2. extraer los ficheros y directorios (con su jerarquía de ficheros y directorios) de un archivo tar.

   Sintaxis: tar -[v]xf nombre_fichero.tar

   Opciones:

   - **f** (read): este modificador debe situarse siempre al final de la lista de modificadores e indica que se utilice el nombre_fichero.tar como archivo para desempaquetar. Si no se emplea esta opción, el valor predeterminado para recuperar es /dev/mto (cinta magnética), dispositivo que puede no estar configurado en la máquina o al que no siempre se tiene permiso para su uso.
   - **x** (extract): extrae los archivos/directorios incluidos en el fichero tar.
   - **v** (verboso): sirve para que se visualicen los archivos y directorios que se van a recuperar.

   Cuando se desempaqueta un archivo tar (opción –x) es posible que se sobrescriban ficheros o directorios que tengan el mismo nombre que los que se están extrayendo, por lo que normalmente se crea previamente un directorio destinado a tal fin.

```
Ejemplos:
       tar -cvf ficheros.tar fich1 fich2 fich3
       tar -cvf ficheros1.tar fichero*
       tar -xvf ficheros.tar
```

5. COMUNICACIÓN CON OTROS USUARIOS

Linux ofrece un sistema operativo **multiusuario** y por lo tanto dispone de comandos que permiten que estos se comuniquen entre sí. Entre dichos comandos tenemos:

write. Envía un mensaje

Esta utilidad permite a un usuario enviar un mensaje a otro que está conectado al sistema, mostrándole dicho mensaje como un aviso en el terminal del otro usuario.

Sintaxis: write nombre_usuario [Terminal]

El nombre de usuario es el login con el que se desea comunicar, y el terminal (que es opcional) es útil ponerlo si el usuario ha iniciado más de una sesión.

Para finalizar la conversación se debe pulsar Control+D.

```
Ejemplo:
       write mario
```

mesg. Deniega o acepta mensajes

Este comando sirve para denegar o aceptar la recepción de mensajes procedentes de otros usuarios.

Sintaxis: mesg [n / y]

Con la opción n no se aceptan mensajes procedentes de otros usuarios y con la opción y si se aceptan.

Si se ejecuta el comando sin opciones se puede comprobar cómo tenemos la recepción de mensajes. Si nos muestra el mensaje *is y* si aceptamos los mensajes y si es *is n* entonces no se aceptan.

```
Ejemplo:
Si queremos desactivar la recepción de mensajes
     mesg n
```

6. COMPILACIÓN Y EJECUCIÓN DE PROGRAMAS ESCRITOS EN C/C++

Un programa en C/C++ se escribe mediante un editor (por ejemplo: vi, pico, etc). Si el fichero contiene un programa escrito en C su extensión será **.c,** y si contiene un programa escrito en C++ su extensión será **.cpp**.

Para compilar un programa escrito en C se usa el compilador de C **gcc**, cuya sintaxis es:

gcc nombre_fichero_programa (con la extensión .c)

dando como resultado un fichero denominado ***a.out*** que será el ejecutable. Si se quiere que el ejecutable tenga otro nombre distinto se usará la opción **-o** del compilador de la siguiente forma:

gcc –o nombre_fichero_ejecutable nombre_fichero_programa (con la extensión .c)

Para compilar un programa escrito en C++ se usa el compilador de C++ **g++**, cuya sintaxis es:

g++ nombre_fichero_programa (con la extensión .cpp)

dando como resultado un fichero denominado ***a.out*** que será el ejecutable. Si se quiere que el ejecutable tenga otro nombre distinto se usará la opción **-o** del compilador de la siguiente forma:

g++ –o nombre_fichero_ejecutable nombre_fichero_programa (con la extensión .cpp)

Ninguno de estos dos compiladores se detiene al encontrar el primer error, sino que continúa mientras puede.

El fichero ejecutable obtenido se ejecuta invocándolo desde la línea de comandos:

$ a.out (o nombre_fichero_ejecutable)

es posible que sea necesario especificar que dicho fichero está en el directorio de trabajo:

$ ./a.out (o nombre_fichero_ejecutable)

```
Ejemplo:
Se desea visualizar por pantalla un número concreto de números aleatorios. Los pasos a seguir
para implementar y ejecutar un programa en C que lleve a cabo esta tarea podrían ser los
siguientes:

1.  Usando un editor (por ejemplo, vi) se escribe el programa y se almacena en un fichero con
    extensión .c
        $vi  aleatorio.c
                #include <stdio.h>
                #include <stdlib.h>

                int main ()
```

```
{
        int num;  // Cantidad de numeros aleatorios
        int i;   // Controla el ciclo

        printf ("Cuantos numeros aleatorios deseas?");
        scanf ("%d", &num);

        for (i=0; i<num; i++)
        {
                printf ("Numero %d aleatorio: %d \n", i, rand());
        }

        return 0;
}
```

2. Se compila usando el compilador gcc, y si por ejemplo, se desea que su ejecutable se almacene en el fichero aleatorio.out

```
$gcc  -o  aleatorio.out  aleatorio.c
```

3. Por último, se ejecuta.

```
$./aleatorio.out
```

7. GESTIÓN DE TERMINALES

screen. Permite multiplexar terminales

Esta utilidad permite a un usuario cuando está trabajando en un terminal, pasar esa ventana a segundo plano y seguir trabajando en el mismo terminal. De la misma forma, cuando el usuario lo necesite, podrá pasar de nuevo esa ventana a un primer plano y continuar trabajando en ella.

Sintaxis: screen [-ls]
screen [-r] numero_sesion

Opciones:

- La opción **–ls** sirve para mostrar la lista de ventanas en segundo plano.
- La opción **-r** sirve para pasar a primer plano una ventana, indicándole el número de sesión correspondiente.

Usando el comando **exit** se cierra la ventana que en ese momento este activa.
Para poder usar este comando es necesaria la instalación del mismo.

Ejemplo:

1. Se ejecuta el comando para pasar la ventana actual a un segundo plano

```
screen
```

2. Se consulta el listado de ventanas en segundo plano

```
screen –ls
There is a screen on:
        13022.pts-0.singgroup-warc-pc   (18/09/20 08:26:00)     (Attached)
1 Socket in /run/screen/S-sing-group.
```

3. Se vuelve a primer plano

```
screen –r 13022
```

Nota: Existe otro multiplexor de terminales equivalente a **screen** que es **tmux**.