Capítulo 8: Administración de procesos

Introducción

El monitoreo de procesos en Linux es una tarea sumamente crítica para los administradores de sistemas. Esto incluye tanto el identificar como el manipular procesos. En este último aspecto es fundamental poder modificar la cantidad de recursos que se le otorga a cada proceso (concepto conocido como *prioridad*) de acuerdo a lo que se necesite, y también alterar su funcionamiento mediante el envío de señales cuando la situación lo amerite. No menos es importante es la necesidad de reiniciar servicios luego de haber modificado archivos de configuración a fin de aplicar los cambios al funcionamiento de estos.

Definición y tipos de procesos

Los procesos en Linux son ni más ni menos que programas que están corriendo en un momento dado. Algunos de ellos inician a su vez otros procesos, y así sucesivamente en algunos casos. Cuando un proceso **A** inicia otro proceso **B**, se dice que **A** es el padre de **B** (o lo que es lo mismo, que **B** es hijo de **A**).

Dependiendo de la forma en que corren a estos programas los podemos clasificar en tres grandes categorías:

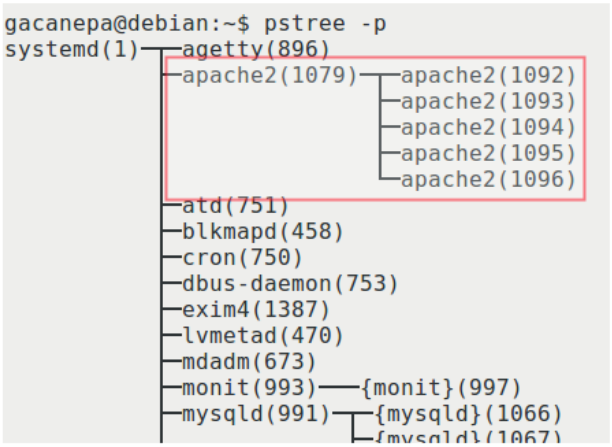
- Los **procesos normales** generalmente son ejecutados en una terminal y corren en el sistema operativo a nombre de un usuario.
- Los **procesos daemon** también se ejecutan a nombre de un usuario pero no tienen salida directa por una terminal, sino que corren en segundo plano. Generalmente los conocemos como **servicios**, y en vez de utilizar una terminal para *escuchar* por un requerimiento, lo hacen a través de un *puerto*.
- Los **procesos zombies** son aquellos que han completado su ejecución pero aún tienen una entrada en la tabla de procesos debido a que no han enviado la señal de finalización a su proceso padre (o este último no ha podido leerla). Usualmente la presencia de este tipo de procesos indica un error en el diseño del programa involucrado. Muchos procesos zombies pueden llegar a ocasionar que tengamos que reiniciar el equipo debido a que no queden identificadores de proceso disponibles para asignar.

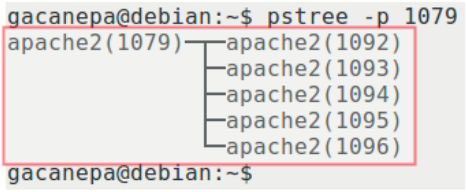
En nuestro sistema operativo el árbol de procesos está representado en el directorio **/proc**, que es una estructura de árbol virtual que genera y monta nuestro kernel durante el arranque. Para examinar el contenido de este directorio y poder ver el estado de los procesos en un formato amigable para nosotros disponemos de varios comandos que presentamos a continuación.

El comando pstree

Esta herramienta nos permite observar la relación entre procesos padre y sus hijos al visualizar los procesos de nuestro sistema y la relación que existe entre ellos. Si la utilizamos con la opción `-p`, podremos ver el **P**rocess **ID**entifier (o **identificador de proceso**, comúnmente llamado **PID**) de cada proceso. En este contexto, decimos que si el proceso **A** es el padre de **B**, el **PID** de **A** recibe el nombre de **PPID** (**P**arent **P**rocess **ID**) de **B**.

Si a continuación de la opción `-p` indicamos un PID dado, el árbol de procesos se mostrará comenzando por el proceso al que le corresponde dicho PID:





En resumen, podemos decir que el comando `pstree` nos devuelve una lista de procesos en ejecución en vista de *árbol* (de forma jerárquica).

El comando ps

A pesar de lo útil que es `pstree`, el comando más conocido y utilizado para listar procesos es `ps` por la gran cantidad de opciones que tiene disponibles. El uso de `-ef` o `aux` (esta última sin guión medio al comienzo) hace que podamos visualizar rápidamente -con distinto grado de detalle- el listado de procesos que están corriendo en nuestro sistema. Una característica distintiva de `ps` es que no es interactivo, sino que saca una *foto* de los procesos que están corriendo en un determinado momento. Sin opciones, `ps` nos devuelve los procesos ligados a la terminal actual.

El uso de las opciones a, u, y x de manera separada devuelve una lista de 1) todos los procesos que se están ejecutando en una terminal, 2) muestra el estado de los procesos del usuario actual y qué cantidad de recursos está requiriendo cada uno, y 3) indica la información de los

demonios y procesos sin terminal. Al combinar estas opciones como aux podemos acceder a todos esos datos simultáneamente.

En la imagen siguiente vemos la salida de `ps aux` y de cada una de sus opciones utilizada por separado. También observamos el resultado de `ps -ef`. Por cuestiones de espacio, hemos enviado la salida de los comandos a una tubería (|) de manera que `head -n 2` a continuación muestre las dos primeras líneas de cada caso. Esto es suficiente para ilustrar el uso del comando y para inspeccionar el significado de las columnas en el resultado:

```
gacanepa@debian:~$ ps aux | head -n 2
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.2  1.3 139220  6892 ?        Ss   19:57   0:00 /sbin/init
gacanepa@debian:~$
gacanepa@debian:~$ ps a | head -n 2
  PID TTY      STAT   TIME COMMAND
  896 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
gacanepa@debian:~$
gacanepa@debian:~$ ps u | head -n 2
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
gacanepa  1402  0.0  0.9  21248  4952 pts/0    Ss   19:58   0:00 -bash
gacanepa@debian:~$
gacanepa@debian:~$ ps x | head -n 2
  PID TTY      STAT   TIME COMMAND
 1394 ?        Ss     0:00 /lib/systemd/systemd --user
gacanepa@debian:~$
gacanepa@debian:~$ ps -ef | head -n 2
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 19:57 ?        00:00:00 /sbin/init
gacanepa@debian:~$
```

Veamos el significado de cada columna:

- **USER**: usuario dueño del proceso.
- **PID**.
- **%CPU**: porcentaje de tiempo de CPU utilizado sobre el tiempo que el proceso ha estado en ejecución.
- **%MEM**: porcentaje de memoria física utilizada.
- **VSZ**: memoria virtual del proceso medida en KiB.
- **RSS** (**R**esident **S**et **S**ize) es la cantidad de memoria física no *swappeada* que la tarea ha utilizado (en KiB)
- **TT**: terminal asociada al proceso. Si en este campo vemos un signo de pregunta (**?**) significa que el proceso en cuestión se trata de un servicio que no está utilizando ninguna terminal.

- **STAT**: código de estado. Una **R** en este campo indica que el proceso se está ejecutando. Una **S** nos dice que está *durmiendo* esperando a que suceda un evento.
- **STARTED**: fecha y hora de inicio del proceso.
- **TIME**: tiempo de CPU acumulado.
- **COMMAND**: comando relacionado con el proceso, incluyendo todos sus argumentos.

Es importante notar que `ps` nos permite adaptar la cantidad y el orden de las columnas a mostrar, e inclusive nos deja ordenar el resultado utilizando una de ellas de manera ascendente o descendente. Eso es posible mediante las opciones `-eo` y `--sort`, respectivamente. La primera debe ser seguida por la lista de campos a mostrar (en el ejemplo que aparece a continuación utilizamos PID, PPID, comando, uso de memoria, y uso de CPU), mientras que a continuación de la segunda debemos colocar un signo igual y el criterio de ordenamiento (`-%mem` indica ordenar por uso de memoria en forma descendente):

```
ps -eo pid,ppid,cmd,%cpu,%mem --sort=-%mem
```

```
gacanepa@debian:~$ ps -eo pid,ppid,cmd,%cpu,%mem --sort=-%mem
  PID  PPID CMD                         %CPU %MEM
  991     1 /usr/sbin/mysqld             0.0 14.5
 1079     1 /usr/sbin/apache2 -k start   0.0  5.1
  846     1 php-fpm: master process (/e  0.0  4.8
 1106     1 /usr/sbin/smbd               0.0  3.0
 1423  1106 /usr/sbin/smbd               0.0  2.6
    1     0 /sbin/init                   0.0  1.3
 1392   874 sshd: gacanepa [priv]        0.0  1.3
 1092  1079 /usr/sbin/apache2 -k start   0.0  1.2
 1093  1079 /usr/sbin/apache2 -k start   0.0  1.2
 1094  1079 /usr/sbin/apache2 -k start   0.0  1.2
 1095  1079 /usr/sbin/apache2 -k start   0.0  1.2
```

Los campos que podemos incluir a continuación de la opción combinada `-eo` se describen en la sección **STANDARD FORMAT SPECIFIERS** en el *man page* de `ps`.

Los comandos kill y killall

Hay ocasiones en que un administrador del sistema puede desear interrumpir la normal ejecución de un proceso, ya sea debido a que olvidó incluir un parámetro en el comando que lo inició, porque no se está comportando de la manera esperada, o bien porque está impactando negativamente en el funcionamiento del sistema. En Linux generalmente hacemos referencia a esta acción como *matar procesos*, y ahora veremos las maneras más usuales de llevarla a cabo.

Para evitar errores, antes de matar procesos es preferible identificar aquellos que se verían afectados por nuestra medida. Por ejemplo, antes de matar el proceso con PID 3092 es una buena idea identificarlo primero mediante `ps` y la opción `--pid`. La razón de esto es asegurarnos de que apuntamos correctamente el PID a fin de no interrumpir el proceso equivocado:

```
ps --pid 3092
```

Si la descripción del proceso corresponde con lo esperado, podemos proceder a matarlo con el comando `kill` seguido del PID del proceso en cuestión:

```
kill 3092
```

Si quisiéramos interrumpir todos aquellos que tienen un PPID en común (o todos los que pertenezcan a un usuario o grupo en particular) ir uno por uno sería tedioso. Por tal motivo, disponemos de los comandos `pgrep` y `pkill` para identificar y matar varios procesos de una sola vez.

A manera de ilustración, identifiquemos aquellos procesos que comparten el padre cuyo PID es 1576:

```
pgrep -l -P 1576
```

Antes de reemplazar `pgrep` por `pkill`, preguntémonos si realmente deseamos matar todos esos procesos. Si la respuesta es sí, podemos hacer

```
pkill -P 1576
```

Si en vez de realizar la identificación por PPID quisiéramos hacerlo a partir del propietario o del grupo dueño del proceso, simplemente deberemos reemplazar la opción `-P` con `-u` o `-G`, respectivamente, seguido del nombre o del identificador del usuario o grupo.

En el caso de que existan varios procesos que compartan el mismo nombre (pero con PID distintos, por supuesto) y deseemos detenerlos a todos, podemos hacer uso del comando `killall`, cuya sintaxis es idéntica a `kill`. Esto puede suceder en el caso del servidor Apache y sus procesos relacionados, por ejemplo, donde todos comparten el nombre **apache2** (Debian y derivados) o **httpd** (CentOS y similares).

Detrás de escenas: envío de señales a procesos

Cuando empleamos `kill` para matar un proceso, se le envía al mismo una señal. En otras palabras, le estamos mandando una indicación para modificar su funcionamiento normal. Los distintos tipos de señales se enumeran a través de

```
kill -l
```

```
gacanepa@debian:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
gacanepa@debian:~$
```

Las señales más utilizadas son **1** (**SIGHUP**), **9** (**SIGKILL**), y **15** (**SIGTERM**):

- Cuando se cierra la terminal asociada a uno o más procesos, se envía a los mismos la señal **SIGHUP**, lo que hace que se detengan.
- **SIGKILL** le indica a un proceso que debe finalizar de inmediato, sin darle tiempo a liberar adecuadamente los recursos que esté utilizando. Esta señal no puede ser ignorada por el proceso al que es enviada. Por ejemplo, cualquiera de los siguientes comandos enviará un **SIGKILL** al proceso cuyo PID es 964:

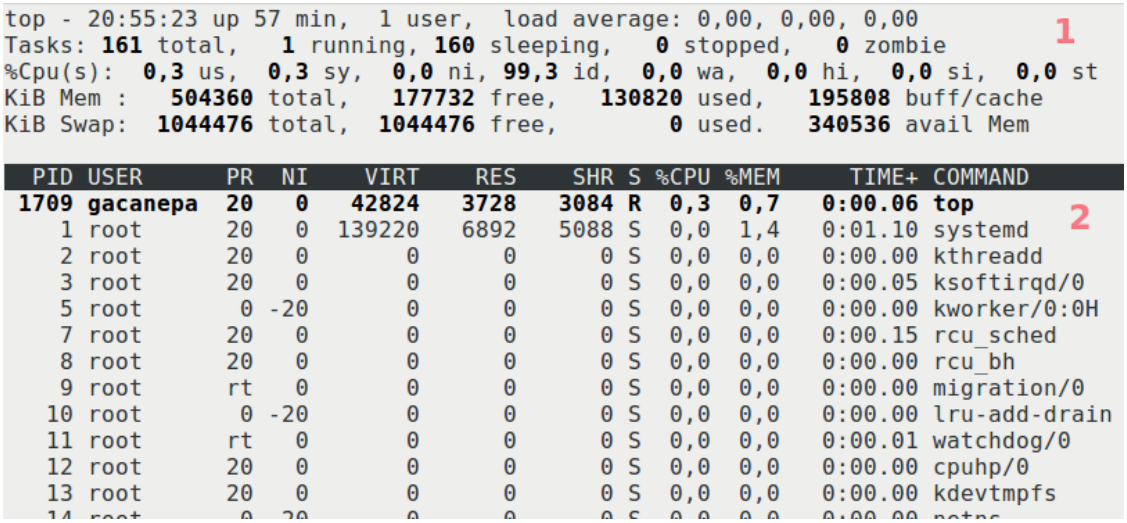
  kill -9 964 kill -s SIGKILL 964

- Finalmente, **SIGTERM** le permite al proceso terminar su ejecución normalmente dándole la oportunidad de liberar los recursos utilizados. Aunque esto suene bien, cabe aclarar que esta señal puede ser ignorada por un proceso. Por eso puede ser necesario (como último recurso) utilizar **SIGKILL** en algunas ocasiones.

El comando top

Una de las limitaciones de `ps` es que devuelve una *foto* del estado de los procesos en el momento que el comando se ejecuta. En cambio, `top` actualiza la información cada cierto tiempo (por defecto, cada tres segundos) y además provee otros datos útiles sobre el estado del sistema.

En la siguiente imagen vemos a `top` en funcionamiento. En la pantalla podemos distinguir dos áreas principales, marcadas con **1** y **2**.



En las dos primeras filas del área marcada con el 1 vemos información sumamente importante sobre el estado actual del sistema:

Fila 1:

- **top** es el nombre del programa.
- **20:55:23** es la hora actual.
- **up 57 min** indica que el equipo ha estado encendido por 57 minutos.
- **load average: 0.00, 0.00, 0.00** muestra la cantidad de procesos (en promedio) que están utilizando (o han estado esperando para utilizar) el CPU durante los últimos 60 segundos, 5 minutos, y 15 minutos, respectivamente.

Para ilustrar el significado de los valores del *load average* (también llamado *carga promedio*), supongamos que los valores sean diferentes (digamos 0.64, 0.30, 0.69, por ejemplo). En este supuesto, 0.64 significa que durante los últimos 60 segundos el CPU ha estado *desocupado* el 36% del tiempo. De la misma manera, 0.30 y 0.69 indican que durante los últimos 5 y 15 minutos el CPU ha estado *desocupado* el 70% y el 31% del tiempo, respectivamente.

Fila 2:

- **Tasks: 161 total, 1 running, 160 sleeping, 0 stopped, 0 zombie** muestra la cantidad de procesos que están corriendo actualmente en el sistema y sus posibles estados:
- *running* indica la cantidad de procesos que están corriendo. En la sección indicada con el 2 se los identifica con la letra **R** en la columna **S** (de **S**tatus).
- *sleeping* representa el número de procesos que no están corriendo actualmente pero que se encuentran esperando que ocurra un evento para despertarse (por

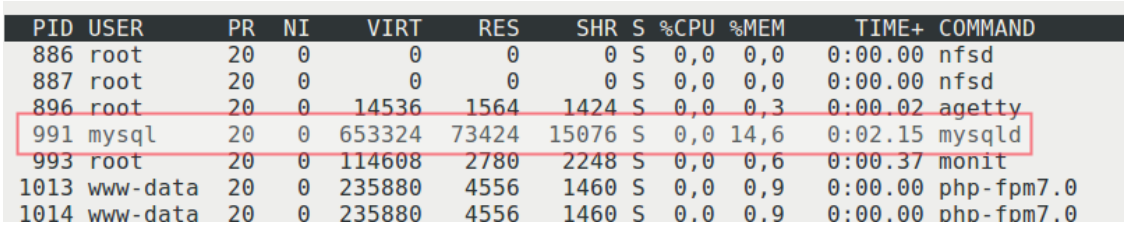
ejemplo, una consulta al servidor web). Se indican con la letra **S** en la misma columna mencionada anteriormente.

- *stopped* son aquellos procesos que han sido detenidos. Se identifican con la letra **T**.
- Los procesos zombie se indican con la letra **Z**.

En el área marcada con el 2 podemos apreciar los siguientes datos sobre cada proceso (de izquierda a derecha):

- **PID**: Process ID.
- **USER**: Usuario dueño del proceso.
- **PR**: Prioridad de ejecución del proceso.
- **NI**: Es un valor que refleja la prioridad sobre el uso de los recursos del sistema otorgada a cada proceso en particular.
- **VIRT**: Cantidad de memoria virtual que el proceso está manejando.
- **RES**: Es la representación más cercana a la cantidad de memoria física que un proceso está utilizando.
- **SHR**: La cantidad de memoria compartida (potencialmente con otros procesos) disponible para este proceso en particular, expresada en KiB.
- **S**: Estado del proceso.
- **%CPU**: Es el porcentaje de tiempo de CPU utilizado por el proceso desde el último refresco de pantalla.
- **%MEM**: Indica el porcentaje correspondiente al uso de memoria física de un proceso en particular.
- **TIME+**: Tiempo de CPU que ha utilizado el proceso desde que inició, expresado en *minutos:segundos.centésimas* de segundo.
- **COMMAND**: Muestra el comando que se utilizó para iniciar el proceso o el nombre de este último.

Veamos otro ejemplo:



```
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  886 root      20   0       0      0      0 S  0,0  0,0   0:00.00 nfsd
  887 root      20   0       0      0      0 S  0,0  0,0   0:00.00 nfsd
  896 root      20   0   14536   1564   1424 S  0,0  0,3   0:00.02 agetty
  991 mysql     20   0  653324  73424  15076 S  0,0 14,6   0:02.15 mysqld
  993 root      20   0  114608   2780   2248 S  0,0  0,6   0:00.37 monit
 1013 www-data  20   0  235880   4556   1460 S  0,0  0,9   0:00.00 php-fpm7.0
 1014 www-data  20   0  235880   4556   1460 S  0,0  0,9   0:00.00 php-fpm7.0
```

Consideremos el proceso con PID 991 (**mysqld**). Entre otras cosas, podemos destacar que dicho proceso está siendo ejecutado como el usuario **mysql** (USER), está utilizando alrededor de 73424 KiB de RAM (RES), lo cual corresponde al **14.6%** del total de memoria instalada en el equipo (%MEM). A comparación del resto de los procesos, está ocupando bastante más tiempo de CPU (2 minutos, 15 segundos).

Como toda herramienta de la línea de comandos, top posee numerosas opciones. Entre las más útiles podemos destacar las siguientes:

- Refrescar la pantalla cada X segundos: `top -d X`
- Ordenar la salida por uso de RAM (descendente): `top -o %MEM`
- Monitorear sólo los procesos cuyos PIDs son 324, 697, y 25216: `top -p 324,697,25216`

Como siempre, podemos buscar más detalles sobre el uso de esta herramienta al examinar su *man page*.

Los comandos nice y renice

Ya hemos visto que en un momento dado puede haber numerosos procesos corriendo en nuestro sistema (al menos esa es la percepción que tenemos). Probablemente algunos de ellos sean más críticos que otros y por lo tanto deberán poder utilizar los recursos del sistema cuando los necesiten. Esto se logra estableciendo la prioridad de ejecución de procesos, algo que mencionamos en la sección anterior.

Las columnas **PR** y **NI** en la salida de `top` indican la prioridad que se le ha asignado a un proceso en particular -tal como lo ve el kernel en un momento dado- y el valor de *niceness* del mismo. Este último valor puede considerarse como una especie de *pista* que se le da al kernel para indicar la prioridad que debe darse inicialmente al proceso. **PR** y **NI** están relacionados entre sí: mientras **mayor** sea la prioridad de un proceso, es **menos *nice*** (del inglés ***bueno***) ya que consumirá más recursos del sistema. El valor de *niceness* puede ubicarse en algún lugar del intervalo comprendido entre -20 y 19. Estos dos límites representan la mayor y la menor prioridad posibles, respectivamente.

Para modificar la prioridad de un proceso en ejecución, utilizaremos el comando `renice`. Los usuarios con privilegios limitados solamente pueden *aumentar* el valor de niceness correspondientes a un proceso del cual son dueños, mientras que root puede

modificar este valor para cualquier proceso sin importar quién sea el usuario dueño del mismo.

Por ejemplo:

- Cambiar la prioridad del proceso con PID 324 (opción `-p`) a 10: `renice -n 10 -p 324`
- Cambiar la prioridad de todos los procesos del usuario (opción `-u`) **alumno** a -10: `renice -n -10 -u alumno`

Por defecto, cualquier nuevo proceso se ejecuta con una prioridad igual a 0. Si deseáramos iniciarlo con una prioridad diferente, podemos hacer uso del comando `nice` seguido de la opción `-n`, del nuevo valor de niceness deseado, y del comando a ejecutar.

Por ejemplo, iniciemos `top` con un valor de *niceness* igual a 10:

```
nice -n 10 top
```

El resultado de la ejecución de `nice` y `renice` pueden comprobarse al observar las columnas **PRI** y **NI** en la salida de `top`.

Administración de servicios

Para controlar el inicio y los servicios del sistema, en distribuciones que utilizan **systemd** se dispone de una herramienta llamada `systemctl`. En systemd se utiliza el término unit para referirse no solamente a servicios, sino también a puntos de montaje, sockets, e incluso dispositivos.

A continuación, presentamos las opciones más utilizadas de `systemctl`. En todas ellas hacemos referencia a un servicio ficticio llamado *miservicio*:

- **Iniciar** el servicio: `systemctl start miservicio.service`
- Configurarlo para que **inicie al arrancar el equipo**: `systemctl enable miservicio.service`
- **Detenerlo**: `systemctl stop miservicio.service`
- **Impedir** que inicie al arrancar el equipo: `systemctl disable miservicio.service`
- **Reiniciar** el servicio: `systemctl restart miservicio.service`
- Averiguar si está configurado para arrancar al inicio: `systemctl is-enabled miservicio.service`

- Averiguar si está corriendo actualmente: `systemctl is-active miservicio.service` o `systemctl -l status miservicio.service` (esta última variante provee más información sobre el estado y la operación del servicio).

En los ejemplos anteriores, podemos omitir el `.service` si estamos seguros que no hay otra unit de distinto tipo con el nombre *miservicio*.

Para cambiar el objetivo por defecto a **graphical.target** (entrará en efecto con el próximo reinicio), debemos recurrir a:

```
systemctl set-default graphical.target
```

Cuando es necesario realizar tareas de mantenimiento o de emergencia será necesario que nos cambiemos al modo monousuario. Para llevar a cabo esta acción haremos uso de

```
systemctl rescue
```

o

```
systemctl emergency
```

La diferencia entre **rescue** y **emergency** consiste en que la última es mucho más reducida. La primera provee servicios básicos y monta sistemas de archivos, mientras que la segunda no.

Entradas, salidas, redirecciones, y tuberías

Todos los procesos para poder lanzarse necesitan tener lo que se conoce como *entrada estandar* (más comúnmente llamado por su nombre en inglés **stdin**) y devuelven como resultado dos archivos que son capturados por la terminal en la cual estamos trabajando: la *salida estándar* (**stdout**) y el *error estándar* (**stderr**).

Por ejemplo, cuando se ejecuta el comando `ls`, el mismo toma su **stdin** a partir de lo que escribimos en el teclado. Luego de que presionamos Enter, el sistema operativo procesa el comando y como resultado de ello vemos la salida. Para ilustrar, utilicemos el siguiente comando (donde el directorio **/mi/directorio** no existe):

```
ls /var /mi/directorio
```

En la imagen podemos ver el resultado del comando anterior:

```
gacanepa@debian:~$ ls /var /mi/directorio
ls: no se puede acceder a '/mi/directorio': No existe el fichero o el directorio
/var:
backups  lib    lock  lost+found  opt  spool  www
cache    local  log   mail        run  tmp
gacanepa@debian:~$
```

El mensaje *ls: no se puede acceder a '/mi/directorio': No existe el fichero o el directorio* forma parte de **stderr**, mientras que los contenidos de **/var** representan **stdout**.

Gráficamente, el procedimiento anterior puede describirse de la siguiente manera:

Como las salidas **stdout** y **stderr** son archivos, al fin y al cabo, los podemos trabajar como tales e incluso capturarlos por la terminal o (y esto es lo más interesante) también pueden direccionarse a otros archivos en disco para su posterior inspección. Para ello vamos a usar el símbolo **>**.

Volviendo a utilizar el ejemplo anterior, para guardar **solamente stdout** en el archivo salida.txt haríamos lo siguiente:

```
ls /var /mi/directorio > salida.txt
```

Si deseamos almacenar únicamente stderr:

```
ls /var /mi/directorio 2> salida.txt
```

Para guardar ambos:

```
ls /var /mi/directorio 2>&1 salida.txt
```

Si ejecutamos cualquiera de estos comandos en repetidas ocasiones veremos que solamente nos guarda en el archivo la **última stdout** o **stderr**. Para agregar contenido al archivo debemos usar el operador **>>**.

Existe además otra forma de trabajar con las salidas, y es transformar la salida **stdout** de un comando en **stdin** de otro. Para eso utilizamos el símbolo **|**, conocido como **pipe** o **tubería**.

Como ya sabemos, el comando `ps aux` nos devolverá un listado de todos los procesos en ejecución. Si en vez de desear mostrar esa salida por pantalla la enviamos como entrada del comando `grep`, podemos filtrarla y mostrar solamente las líneas que cumplan un determinado patrón.

```
ps aux | grep apache
ps aux | grep apache | grep -v grep
```

```
gacanepa@debian:~$ ps aux | grep apache
root      1073  0.0  5.6 287752 28728 ?        Ss   08:53   0:00 /usr/sbin/apache2 -k start
webserv+  1524  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
webserv+  1525  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
webserv+  1526  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
webserv+  1527  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
webserv+  1528  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
gacanepa  1675  0.0  0.1  12784   960 pts/0    S+   09:16   0:00 grep apache
gacanepa@debian:~$ ps aux | grep apache | grep -v grep
root      1073  0.0  5.6 287752 28728 ?        Ss   08:53   0:00 /usr/sbin/apache2 -k start
webserv+  1524  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
webserv+  1525  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
webserv+  1526  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
webserv+  1527  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
webserv+  1528  0.0  1.2 287776  6220 ?        S    08:58   0:00 /usr/sbin/apache2 -k start
gacanepa@debian:~$
```

Si prestamos atención a la imagen anterior, la última línea muestra parte del comando propiamente dicho. Si deseamos eliminar esa línea de la salida de tal manera que veamos únicamente los procesos relacionados con Apache, podemos agregar una tubería adicional como también se muestra en el segundo comando.

Otro ejemplo que nos permitirá ilustrar el uso de las redirecciones y tuberías utiliza el comando `cpio`. Aunque hoy en día el estándar para empaquetar archivos es `tar`, en los primeros días de Unix el uso de `cpio` era el preferido. La razón detrás de esto era la portabilidad de este último por encima del primero, aunque esta situación se revirtió a fines de la década de 1980. A partir de ahí, `tar` se convirtió en la herramienta más utilizada para empaquetar archivos. Sin embargo, saber utilizar `cpio` puede resultar todavía útil si tenemos que lidiar con archivos provenientes de diferentes tipos de sistemas.

Una característica propia a tener en cuenta cuando utilicemos `cpio` es que debemos indicarle la lista de archivos a empaquetar de manera específica. Podemos hacerlo con un comodín (*) o enviándole la salida de `find` o `ls` mediante una tubería como veremos a continuación:

- Empaquetar los archivos de log del servidor web Apache:

  ```
  find /var/log/apache2/ | cpio -o > /tmp/apache2.cpio
  ```

- Ver el contenido de un archivo empaquetado:

  ```
  cpio -tv < /tmp/apache2.cpio
  ```

- Desempaquetar un archivo y restaurar sus contenidos a su ubicación original. Si se desea sobreescribir archivos existentes, utilizar la opción `-u` también:

  ```
  cpio -vid < /tmp/apache2conf.cpio
  ```

- Desempaquetar y restaurar en el directorio actual:

  ```
  cpio -ivd --no-absolute-filenames < /tmp/apache2conf.cpio
  ```

Ahora ya contamos con una herramienta más para empaquetar archivos. Las opciones más utilizadas de `cpio` son:

- Crear un nuevo archivo: `-o` (output)
- Extraer desde un archivo: `-i` (input)
- Copiar una estructura de directorios: `-p` (copypass)
- Reiniciar el tiempo de acceso de los archivos una vez copiados: `-a`
- Preservar tiempos de modificación (usado con `-i`): `-m`
- Crear directorios en caso de que hiciesen falta (usado con `-o` y `-i`): `-d`
- Especificar un archivo que contiene un patrón (usado con `-i`): `-e`
- Listar el contenido de un archivo (usado con `-i`): `-t`
- Indicar el formato del archivo resultante: `-F`

Procesos en primer y segundo plano

En algunos casos, un script puede tomar un tiempo considerable en completar su ejecución, o un programa puede ocupar la línea de comandos mientras se encuentre corriendo. Para permitirnos volver a tomar el control de la terminal en cuestión, la shell nos permite colocar procesos en segundo plano. También podemos iniciar directamente un proceso en segundo plano, de manera que no ocupe la terminal mientras corre.

Para que un proceso se inicie en segundo plano, colocaremos el símbolo & al fin del mismo. Por ejemplo, si queremos correr `updatedb` en segundo plano, podemos hacer:

```
updatedb &
```

Al iniciar un proceso en segundo plano, la terminal queda libre para seguir trabajando en la misma y se nos provee la identificación de dicho proceso y su PID.

Si deseamos traer un proceso a primer plano, nos valdremos el comando `fg` seguido del identificador de este. Si sucediera que no supiéramos el identificador del proceso en cuestión, podemos valernos del comando `jobs` para averiguarlo. Supongamos que

queremos encontrar todos los archivos con la extensión **.sh** y guardar la lista de los mismos en **listadescripts.txt**:

```
find . -iname "*.sh" > listadescripts.txt &
```

Mientras dura la búsqueda, pondremos el comando anterior en segundo plano, lo mostraremos con `jobs`, y lo traeremos de nuevo a primer plano con `fg` seguido del identificador (1 en este caso):

```
root@debian:~# find / -iname "*.sh" > listadescripts.txt &
[1] 1448
root@debian:~# jobs
[1]+  Ejecutando              find / -iname "*.sh" > listadescripts.txt &
root@debian:~# fg 1
find / -iname "*.sh" > listadescripts.txt
root@debian:~#
```

Si el proceso se hubiera iniciado en primer plano, podemos pausar su ejecución con Ctrl + z y luego utilizar el comando bg para enviarlo a segundo plano. De ahí en más podemos proceder como explicamos anteriormente para traerlo nuevamente a primer plano si así lo deseamos.

```
root@debian:~# find / -iname "*.sh" > listadescripts.txt
^Z ⟶ Ctrl + z
[1]+  Detenido                find / -iname "*.sh" > listadescripts.txt
root@debian:~# bg
[1]+ find / -iname "*.sh" > listadescripts.txt &
root@debian:~# jobs
[1]+  Hecho                   find / -iname "*.sh" > listadescripts.txt
root@debian:~# fg 1
-su: fg: 1: no existe ese trabajo
root@debian:~#
```

En el ejemplo de arriba podemos ver que el proceso finalizó antes de que pudiéramos traerlo nuevamente a primer plano.

El comando nohup

La ejecución de un proceso se verá interrumpida si cerramos la terminal desde la que lo iniciamos, o si cerramos la sesión actual. Para evitar que esto suceda, podamos usar nohup, una herramienta que permite iniciar un proceso que sea inmune a la situación que describimos anteriormente.

Poder utilizar nohup es especialmente útil cuando estamos conectados a un equipo de manera remota y deseamos ejecutar un script que continúe su ejecución luego de que cerremos la terminal asociada. De

esta manera, no tenemos que estar logueados durante todo el tiempo que corra el script.

La sintaxis del uso de esta herramienta es la siguiente:

```
nohup [programa a iniciar] &
```

Como podemos ver, generalmente nohup se utiliza en conjunto con el símbolo **&** para enviar simultáneamente el programa en cuestión a segundo plano, aunque no es estrictamente necesario que lo hagamos de esa manera.

Veamos un ejemplo dividido en pasos:

- **Paso 1** – Iniciar un ping a carreralinux.com.ar:

nohup ping carreralinux.com.ar

- **Paso 2** – Desde otra terminal, tomamos nota del PID del comando anterior (1486) y de su padre (1084, el cual corresponde a la primera terminal):

  ps -u gacanepa -o ppid,pid,cmd

- **Paso 3** – Cerramos la primera terminal y volvemos a ejecutar el comando anterior. Vemos que el proceso padre (la shell `bash`) ya no aparece en la lista, y que ahora el padre del ping es el proceso con PID 1.

```
gacanepa@debian:~$ ps -u gacanepa -o ppid,pid,cmd
 PPID   PID CMD
    1  1015 /lib/systemd/systemd --user
 1015  1037 (sd-pam)
  914  1083 sshd: gacanepa@pts/0
 1083  1084 -bash
 1468  1474 sshd: gacanepa@pts/1
 1474  1475 -bash
 1084  1486 ping carreralinux.com.ar
 1475  1487 ps -u gacanepa -o ppid,pid,cmd
gacanepa@debian:~$ ps -u gacanepa -o ppid,pid,cmd
 PPID   PID CMD
    1  1015 /lib/systemd/systemd --user
 1015  1037 (sd-pam)
 1468  1474 sshd: gacanepa@pts/1
 1474  1475 -bash
    1  1486 ping carreralinux.com.ar
 1475  1488 ps -u gacanepa -o ppid,pid,cmd
gacanepa@debian:~$
```

De esta manera podemos ver que aunque el proceso padre desapareció de la tabla de procesos, el hijo *huérfano* fue adoptado por **systemd**.