Los procesos

1. Definición y entorno

Un **proceso** representa un programa en curso de ejecución y, al mismo tiempo, todo su entorno de ejecución (memoria, estado, identificación, propietario, padre...).

Los datos de identificación de un proceso son:

- **Un número de proceso único PID** (*Process ID*): se numera cada proceso Unix con el fin de poder diferenciarlo de los otros. El primer proceso iniciado por el sistema es 1, y se trata de un proceso llamado generalmente init. Se utiliza el PID cuando se trabaja con un proceso. Iniciar 10 veces el mismo programa (mismo nombre) produce 10 PID diferentes.
- **Un número de proceso padre PID** (*Parent Process ID*): cada proceso puede iniciar otros procesos, sus procesos hijos (*child process*). Cada proceso hijo debe contener, entre toda su información, el PID del proceso padre que lo inició. Todos los procesos tienen un PPID salvo el proceso 0, que es un seudoproceso que representa el inicio del sistema (crea el 1 init).
- **Un número de usuario y uno de grupo**: corresponde al UID y al GID de la cuenta de usuario que inicia el proceso. El sistema lo utiliza para determinar, a través de la cuenta, los permisos que el proceso tiene para acceder a los recursos. Los procesos hijos heredan ambas cuentas. En algunos casos (que veremos más adelante) se puede modificar este comportamiento.
- **Duración y prioridad del proceso**: la duración del proceso corresponde al tiempo de ejecución consumido desde la última invocación. En un entorno multitarea, el tiempo de procesador se comparte entre los procesos y no todos tienen la misma prioridad. Los procesos de más alta prioridad son ejecutados primero. Cuando un proceso está inactivo, su prioridad aumenta con el fin de tener la oportunidad de ser ejecutado. Cuando está activo, su prioridad baja con el fin de dejar paso a otro. El planificador de tareas del sistema es el que gestiona las prioridades y los tiempos de ejecución.
- **Directorio de trabajo activo**: tras su inicio, se configura el directorio actual del proceso (desde el cual se inició). Este directorio servirá de base para las rutas

relativas.

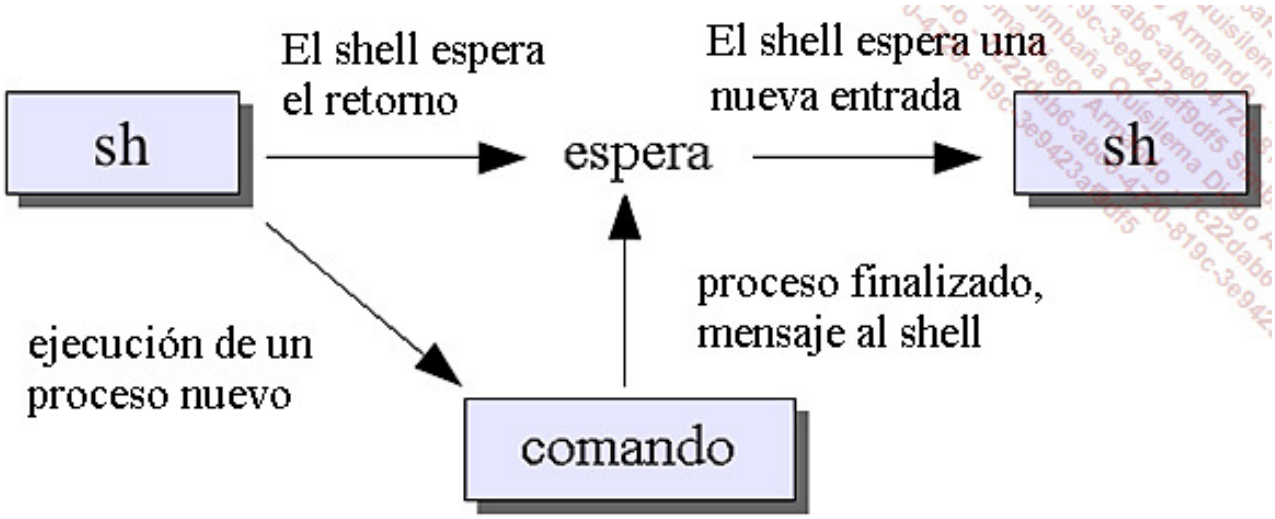
- **Archivos abiertos**: tabla de los descriptores de archivos abiertos. Por defecto al principio sólo hay tres presentes: 0, 1 y 2 (los canales estándar). Con cada apertura de archivo o de nuevo canal, la tabla se rellena. Al cierre del proceso, se cierran los descriptores (en principio).
- Puede encontrar más información, como el tamaño de la memoria asignada, la fecha de inicio del proceso, el terminal de atribución, el estado del proceso, los UID efectivo y real, así como los GID efectivo y real.

2. Estados de un proceso

Durante su vida (tiempo entre el inicio y la salida) un proceso puede pasar por diversos estados o **process state**:

- ejecución en modo usuario (**user mode**);
- ejecución en modo núcleo (**kernel mode**);
- en espera E/S (**waiting**);
- dormido (**sleeping**);
- listo para la ejecución (**runnable**);
- dormido en el swap (**memoria virtual**);
- nuevo proceso;
- fin de proceso (**zombie**).

3. Ejecución en segundo plano



Según lo visto anteriormente, dado que el entorno de trabajo es multitarea, hay un determinado número de procesos que ya se ejecutan en su máquina sin que usted lo vea. Del mismo modo, el shell que utiliza es en sí mismo un proceso. Cuando inserta un comando, el shell crea un nuevo proceso para ejecutarlo, este proceso se convierte en un proceso hijo del shell. Una vez iniciado, hay que esperar al final de su ejecución para iniciar el siguiente (excepto si se utiliza «;» para encadenar los comandos, pero en ese caso, los comandos no se ejecutan al mismo tiempo).

Nada impide al shell esperar el mensaje del proceso terminado para dejar paso: de hecho, una vez iniciado el comando, el shell puede autorizar la inserción de un nuevo comando sin esperar el final de la ejecución del comando anterior. Para ello, basta insertar, después de haber tecleado el comando, el **ampersand "&"**. En este caso, el shell y el comando iniciado funcionarán en paralelo.

```
$ ls -R / > ls.txt 2/dev/null &
[1] 21976
$ [1]   Done            ls -l -R / > ls.txt 2/dev/null
$ ls
fic1      fic3      lista     ls.txt    dir1      users
fic2      fic4      lista2    mypass    pepito.tar.gz
```

Justo después de la inserción, aparece una cifra. Hay que recordarlo, ya que se trata del PID del nuevo proceso iniciado en segundo plano. Después de otra inserción, una línea Done indica que el tratamiento se da por finalizado. El valor [1] es un número de tarea (job).

Algunas observaciones en cuanto al uso del inicio en segundo plano:

- El proceso iniciado en segundo plano no debería esperar ser otro shell, dado que se produciría confusión entre este comando y el propio shell.
- El proceso iniciado no debería mostrar resultados en la pantalla porque entrarían en conflicto con los del shell (por ejemplo, aparición de una línea en medio de una inserción).
- Finalmente, cuando se sale del shell, se sale también de todos sus hijos: en este caso, no abandone el shell en mitad de un procesamiento importante.

4. Background, foreground, jobs

Puede retomar el control del shell si ha iniciado un proceso en segundo plano. Puede pararlo de manera temporal tecleando [Ctrl] **Z**:

```
$ sleep 100
<CTRL+Z>
 [1]+  Stopped                 sleep 100
```

Se ha parado el proceso: se ha suspendido su ejecución hasta que vuelva a ponerlo en primer plano con el comando **fg** (*foreground*):

```
$ fg
sleep 100
```

Cuando ejecuta un comando, obtiene un número entre llaves: es el número de job. Es el mismo número que aparece cuando ejecutamos un comando en segundo plano. Puede obtener una lista de todas las tareas con el comando **jobs**.

```
$ jobs
[1]-  Stopped                 sleep 100
[2]+  Stopped                 sleep 100
```

Los comandos **bg** y **fg** permiten actuar en estos jobs tomando como parámetro su número. Se ejecuta el comando **bg** en un job parado para iniciarlo de nuevo en segundo plano. Se vuelve a iniciar el job 2 en segundo plano:

```
$ bg 2
[2]+ sleep 100 &
$
[2]+  Done                    sleep 100
```

5. Lista de los procesos

El comando **ps** (*process status*) permite obtener información sobre los procesos en curso. Si se ejecuta solo, visualiza únicamente los procesos en curso iniciados por el usuario y desde la consola actual.

```
$ ps
 PID TTY          TIME CMD
4334 pts/1    00:00:00 bash
5017 pts/1    00:00:00 ps
```

Para obtener más información, utilice el parámetro `-f`.

```
ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
seb       4334 24449  0 09:46 pts/1    00:00:00 /bin/bash
seb       5024  4334  0 10:51 pts/1    00:00:00 ps -f
```

El parámetro `-e` da información sobre todos los procesos en curso.

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
...
```

```
seb     26431     1  0 Mar04 ?       00:00:30 kded [kdeinit]
seb     26436 26322  0 Mar04 ?       00:00:03 kwrapper ksmserver
seb     26438     1  0 Mar04 ?       00:00:00 ksmserver [kdeinit]
seb     26439 26424  0 Mar04 ?       00:00:50 kwin [kdeinit
seb     26441     1  0 Mar04 ?       00:00:28 kdesktop [kdeinit]
seb     26443     1  0 Mar04 ?       00:03:40 kicker [kdeinit]
seb     26453     1  0 Mar04 ?       00:00:00 kerry [kdeinit
seb     26454 26424  0 Mar04 ?       00:00:01 evolution
seb     26465 26424  0 Mar04 ?       00:00:11 kde-window-decorator
seb     26467     1  0 Mar04 ?       00:00:02 gconfd-2 12
seb     26474     1  0 Mar04 ?       00:00:01 knotify [kdeinit]
seb     26485     1  0 Mar04 ?       00:03:06 beagled
...
```

El parámetro `-u` permite precisar una lista de uno o varios usuarios separados por una coma. El parámetro `-g` efectúa lo mismo pero para los grupos, `-t` para los terminales y `-p` para unos PID determinados.

```
$ ps -u root
  PID TTY          TIME CMD
    1 ?        00:00:05 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 migration/0
    4 ?        00:00:09 ksoftirqd/0
    5 ?        00:00:23 events/0
    6 ?        00:00:00 khelper
   25 ?        00:00:00 kblockd/0
   26 ?        00:00:00 kacpid
   27 ?        00:00:00 kacpi_notify
  130 ?        00:00:00 cqueue/0
  131 ?        00:00:00 kseriod
  156 ?        00:00:22 kswapd0
  157 ?        00:00:00 aio/0...
```

Finalmente el parámetro `-l` ofrece más información técnica.

```
$ ps -l
```

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  4704 24449  0  75   0 -  1213 wait   pts/3    00:00:00 bash
```

A continuación presentamos el detalle de algunas columnas.

Columna	Definición
UID	User ID, nombre del usuario.
PID	Process ID, número del proceso.
PPID	Parent Process ID, número del proceso padre.
C	Factor de prioridad, cuanto más grande es el valor, más elevada es la prioridad.
STIME	Hora de inicio del proceso.
TTY	Nombre del terminal desde el cual se ejecutó el proceso.
TIME	Duración de tratamiento del proceso.
CMD	Comando ejecutado.
F	Banderas del proceso (sale del ámbito de este libro).
S	Estado del proceso S (sleeping) R (running) Z (zombie).
PRI	Prioridad del proceso.
NI	Nice, incremento para el scheduler.

6. Parada de un proceso/señales

Es posible detener un proceso en primer plano con la combinación de teclas [Ctrl] **Z**. Cuando un proceso se ejecuta en segundo plano, no lo puede parar cualquier combinación de teclas, al menos que se utilice el gestor de jobs con fg y bg. Puede resultar necesario mandarle señales especiales a ese proceso en segundo plano. Para ello, hay que emplear el comando **kill**. A diferencia de lo que parece indicar su nombre, la función de este comando no es obligatoriamente destruir o terminar un proceso (recalcitrante o no), sino mandar señales a los procesos.

```
kill [-l] -Num_señal PID [PID2...]
```

La **señal** es uno de los medios de comunicación entre los procesos. Cuando se manda una señal a un proceso, éste debe interceptarla y reaccionar en consecuencia. Se pueden ignorar algunas señales, pero otras no. Según los Unix, se dispone de un número más o menos importante de señales. Se numeran y nombran las señales, pero cuidado: si bien los nombres suelen ser comunes de un Unix a otro, los números no lo son obligatoriamente. La opción **-l** permite obtener la lista de las señales.

Señal	Función
1 (SIGHUP)	El padre manda Hang Up a todos sus hijos cuando termina.
2 (SIGINT)	Interrupción del proceso pedido (tecla [Supr], [Ctrl] **C**).
3 (SIGQUIT)	Ídem SIGINT, pero con generación de un Core Dump (archivo de depuración).
9 (SIGKILL)	Señal que no se puede ignorar, fuerza el proceso a terminar de manera «expeditiva».
15 (SIGTERM)	Señal mandada por defecto por el comando **kill**. Pide al proceso terminar con normalidad.

```
$ sleep 100&
[1] 5187
$ kill 5187
$
[1]+  Completado        sleep 100
$ sleep 100&
[1] 5194
$ kill -9 5194
[1]+  Proceso parado     sleep 100
```

Existen comandos derivados. Con **pkill** la señal no se envía al PID indicado, sino a los comandos. Estos comandos pueden listarse con **pgrep**, que retornará los PID asociados.

```
$ sleep 100 &
[1] 7167
$ sleep 100 &
[2] 7168
$ sleep 100 &
[3] 7169
$ pgrep sleep
7167
7168
7169
$ pkill sleep
[1]   Completado          sleep 100
[2]-  Completado          sleep 100
[3]+  Completado          sleep 100
```

7. nohup

Cuando se sale del shell (exit, [Ctrl] D...) se manda la señal 1 SIGHUP a los hijos para que terminen ellos también. Cuando se inicia un proceso largo en segundo plano y el usuario quiere salir del shell, entonces se para este proceso y habrá que volver a empezarlo. La manera de evitar esto es iniciar el proceso con el comando **nohup**. En este caso, el proceso iniciado ya no reaccionará a la señal SIGHUP, y por lo tanto podrá salir del shell, el

comando seguirá su ejecución.

Por defecto, los canales de salida y error estándares se redirigen hacia un archivo **nohup.out**, salvo si se precisa la redirección de manera explícita.

```
$ nohup ls -lR / &
10206
$ Sending output to nohup.out
```

Cuando un proceso hijo finaliza, manda una señal SIGCHLD a su proceso padre. Salvo excepción (un proceso padre desvinculado del hijo), el proceso padre debe obtener tantos SIGCHLD como hijos inició o como SIGHUP emitió. Si el padre termina antes que los hijos, éstos se convertirán en procesos zombis: las señales SIGCHLD son enviadas a... nadie. El proceso hijo ha finalizado correctamente, ha muerto, no consume ningún recurso. No se puede matar (ya que está muerto), pero sigue ocupando una entrada en la tabla de los procesos. init lo recupera, y como está siempre en espera, el zombi puede acabar desapareciendo.

8. nice y renice

El comando **nice** permite iniciar un comando con una prioridad más baja para permitir a otros posibles procesos ejecutarse más rápidamente.

```
nice [-valor] comando [argumentos]
```

Un valor positivo causará una bajada de prioridad; uno negativo, el aumento de la prioridad (si está autorizado). El valor debe estar comprendido entre -20 y 20. Cuanto más elevado es el valor, menor es la prioridad.

```
$ nice -10 ls -lR / >lista 2>/dev/null&
10884
$ ps -l
     F S   UID   PID  PPID  C PRI  NI ADDR   SZ WCHAN    TTY
       TIME CMD
80808001 U N+   75  10884  10822 28.5  54  10     0 848K aa3b3a9c ttyp4
       0:03.32 ls
```

El comando **renice** funciona un poco como nice, pero permite modificar la prioridad en función de un usuario, un grupo o un PID. El proceso debe estar ya ejecutándose.

```
renice [-n prio] [-p] [-g] [-u] ID
```

La prioridad debe estar entre -20 y 20. El usuario estándar sólo puede utilizar los valores entre 0 y 20 que permiten bajar la prioridad. La opción **-p** precisa un PID, **-g** un GID y **-u** un UID.

9. time

El comando **time** mide las duraciones de ejecución de un comando, lo que es ideal para conocer los tiempos de ejecución, y devuelve tres valores:

- **real**: duración total de ejecución del comando;
- **user**: duración del tiempo de CPU necesario para ejecutar el programa;
- **system**: duración del tiempo de CPU necesario para ejecutar los comandos relacionados con el OS (llamadas al sistema dentro de un programa).

El resultado aparece por el canal de error estándar 2. Se puede tener una indicación de la carga de la máquina por el cálculo real / (user+system). Si el resultado es inferior a 10, la máquina dispone de un buen rendimiento; más allá de 20, la carga de la máquina es demasiado pesada (rendimiento reducido) o tiene problemas de entrada/salida.

```
$ time ls -lR /home
...
real   4.8
user   0.2
sys    0.5
```

10. exec

Ya hemos visto el comando **exec** cuando hablamos de los canales. Utilizado con un comando como parámetro, remplaza el proceso actual por otro. Su shell por defecto es bash. Si desea cambiarlo por el shell ksh, basta con ejecutar **exec ksh**. He aquí lo que sucede:

```
$ ps | grep bash
 2375 pts/0    00:00:00 bash
$ exec ksh
$
$ ps | grep ksh
 2375 pts/0    00:00:00 ksh
```

Fíjese en los PID asociados a bash y ksh: son los mismos. De esta forma, ksh se ejecuta en el mismo proceso de bash, al que ha reemplazado. Si termina ksh (bash y ksh son muy compatibles, los comandos son los mismos), saldrá de su shell y probablemente será desconectado.

Puede conocer el PID de su shell mostrando el contenido de $$:

```
$ echo $$
2375
```

Veamos la diferencia con un comando interactivo:

```
$ ps
  PID TTY          TIME CMD
 2732 pts/0    00:00:00 bash
 3364 pts/0    00:00:00 ps
[jolivares@client ~]$ ksh
$ ps
  PID TTY          TIME CMD
 2732 pts/0    00:00:00 bash
 3365 pts/0    00:00:00 ksh
 3366 pts/0    00:00:00 ps
```

ksh es un proceso hijo de bash, lleva un PID diferente. Al salir del intérprete de ksh, usted retorna al proceso padre: al prompt de bash.