Introducción a Sistemas Operativos: Procesos

Clips xxx
Francisco J Ballesteros

1. Procesos y nombres

Ya sabes que un proceso es tan sólo una abstracción que implementa el kernel del sistema operativo. Dentro del kernel tenemos un array de records de tal forma que cada record será de tipo `Proc` (o cualquier otro nombre) y definirá el tipo de datos *proceso*.

Como te imaginarás en este punto, cada uno de esos records contiene el path para el directorio actual en que ejecuta el proceso y cualquier otra cosa que UNIX necesite recordar sobre ese proceso. Por ejemplo, qué segmentos de memoria utiliza. Y, naturalmente, los segmentos son también una abstracción y serán tan sólo records que contienen la información que UNIX necesite saber sobre cada uno de ellos. Así de simple.

El nombre de un proceso es parte de la abstracción *proceso* en UNIX. Pero dicho nombre *no* es el nombre del programa que está ejecutando (lo que sería `argv[0]` en la función `main` de dicho programa). El nombre de un proceso es un número entero que identifica el proceso y se conoce como **identificador de proceso** o *process id* o *pid*.

Cuando UNIX crea un proceso le asigna un *pid* que ningún otro proceso ha utilizado antes. Es como un "DNI" para el proceso. Todas las llamadas al sistema que necesitan que indiques sobre qué proceso deben operar reciben un *pid* para que nombres el proceso sobre el que han de trabajar. Pero recuerda que hay muchas llamadas que operan sobre el proceso que hace la llamada y, como es natural, no necesitan que indiques sobre qué proceso han de actuar.

Por ejemplo, cuando un programa llama a `chdir`, UNIX sabe qué proceso está haciendo la llamada (puesto que ha sido UNIX el que lo puso a ejecutar), y dicha llamada opera sobre el proceso en cuestión.

El siguiente programa utiliza la llamada al sistema *getpid(2)* para averiguar el *pid* del proceso que ejecuta el programa.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char* argv[])
{
    pid_t p;

    p = getpid();
    printf("pid is %d\n", p);
    exit(0);
}
```

El tipo `pid_t` es tan solo un tipo de `int`, no hay magia en eso tampoco. Vamos a ejecutar el programa varias veces, tras compilarlo en `pid`:

```
unix$ pid
pid is 91795
unix$ pid
pid is 91800
unix$ pid
pid is 91805
unix$
```

Y cada vez tendrá un nuevo *pid*. En un mismo programa, pero los procesos son distintos.

El comando *ps(1)* lista los procesos que están ejecutando

```
unix$ ps
  PID TT  STAT       TIME COMMAND
15891 p0  Ss      0:00.11 -ksh (ksh)
14610 p0  R+      0:00.00 ps
12349 C0- I       0:00.00 acme
unix$
```

La salida varía mucho de un tipo de UNIX a otro. En este caso el shell que ejecutamos no es `sh`, sino `ksh` y tenemos un editor en el programa `acme` que está ejecutando. Además, podemos ver cómo `ps` está ejecutando también.

Habitualmente `ps` lista sólo los procesos que ejecutan a nombre del mismo usuario que ejecuta `ps`. Pero `ps` tiene muchas opciones y te permite listar todos los procesos que ejecutan en la máquina así como ver muchas otras cosas sobre cada proceso (cuánta memoria virtual está utilizando cada proceso, cuánta memoria real, qué tiempo ha consumido de CPU, cuánto hace que está ejecutando, etc.). Lo mejor es que utilices el manual y recuerdes que en el caso de `ps` las opciones y las columnas que imprime para cada proceso suelen variar de un tipo de UNIX a otro.

Eso sí, lo normal suele ser que se imprima el *pid* de cada proceso (en la primera columna en la mayoría de los UNIX) y el vector de argumentos para cada proceso (normalmente al final de cada línea).

Por ejemplo, en el UNIX que estamos utilizando (un BSD) podemos utilizar estos flags en `ps` para listar todos los procesos y más información sobre cada uno de ellos:

```
unix$ ps -auxw
USER       PID %CPU %MEM   VSZ   RSS TT  STAT  STARTED       TIME COMMAND
elf      15891  0.0  0.0   680   856 p0  Ss    12:48PM    0:00.11 -ksh (ksh)
elf      27386  0.0  0.0   420   452 p0  R+    12:56PM    0:00.00 ps -auw
elf      12349  0.0  0.0   624   692 C0- I     14Jul16    0:00.00 ksh -c /zx/bin/xcmd -s sh -v
elf      26019  0.0  0.0 17720  5040 C0- I     14Jul16    2:36.19 /zx/bin/xcmd
root      5830  0.0  0.0   492  1148 C0  Is+   14Jul16    0:00.01 /usr/libexec/getty ttyC0
root     15869  0.0  0.0   500  1148 C1  Is+   14Jul16    0:00.00 /usr/libexec/getty ttyC1
...
```

Aquí, `VSZ` es la cantidad de memoria virtual y `RSS` es la cantidad de memoria física en uso. La columna `STAT` describe el estado de planificación del proceso (ya sabes... *ejecutando*, *listo para ejecutar*, *bloqueado*). Mira el manual de *ps* para ver qué significan las cosas en la salida que obtengas en tu UNIX.

2. Usuarios

¿A nombre de qué usuario ejecutamos? Veámoslo...

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char* argv[])
{
    int uid;

    uid = getuid();
    printf("uid is %d\n", uid);
    exit(0);
}
```

El nombre de usuario para UNIX es otro entero (como sabes) o *identificador de usuario*, o *uid*. La llamada *getuid(2)* te permite obtener el *uid* del usuario a nombre de quien ejecuta el proceso. Si compilamos el programa en `guid`, podemos ejecutarlo...

```
unix$ guid
uid is 501
unix$
```

Si ejecutamos el comando *id(1)* podemos ver que es correcto:

```
unix$ id
uid=501(nemo) gid=20(staff) groups=20(staff)
unix$
```

En este caso y en este UNIX, mi usuario es el `501`. Igualmente, podemos averiguar a nombre de qué grupo de usuarios está ejecutanto nuestro proceso.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char* argv[])
{
    int gid;

    gid = getgid();
    printf("gid is %d\n", gid);
    exit(0);
}
```

Si compilamos el programa en `ggid`, podemos ejecutarlo...

```
unix$ ggid
gid is 20
unix$
```

¡Ya sabes! Tanto el *uid* como el *gid* son atributos de cada proceso.

¿Y qué nombre de usuario corresponde a cada *uid*? En realidad, estamos adentrándonos en un campo que varía de un UNIX a otro. En general, las cuentas de usuario se abren editando un fichero llamado `/etc/passwd`, y en dicho fichero se suele incluir una línea para cada usuario que detalla su nombre, *uid*, *gid* para cada grupo al que pertenece el usuario, directorio casa, shell que ejecuta el usuario cuando hace un

login, etc. Del mismo modo, el fichero `/etc/group` suele utilizarse para incluir una línea por cada grupo de usuarios con el nombre del grupo y el *gid* del grupo, al menos.

En la mayoría de los UNIX podemos utilizar *getpwuid* para recuperar desde C un record que describe una entrada en *passwd* para un *uid* dado.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>
#include <uuid/uuid.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int uid;
    struct passwd *p;

    uid = getuid();
    p = getpwuid(uid);
    if (p == NULL) {
        err(1, "no passwd for uid");
    }
    printf("user name is %s\n", p->pw_name);
    exit(0);
}
```

Lee *getpwuid(3)* y echa un vistazo a las otras funciones y a los campos de `passwd`. Pero antes vamos a ejecutar este programa:

```
unix$ uidname
user name is nemo
unix$
```

Cualquier usuario puede cambiar su password utilizando *passwd(1)*. Naturalmente el comando *passwd* ejecuta a nombre del usuario que lo ejecuta. La pregunta entonces es... ¿Cómo puede tener *passwd* permisos para cambiar el password? Si tu usuario tiene permisos para editar `/etc/passwd` (o el fichero donde quiera que se guarden los passwords, encriptados) entonces tu usuario tendría acceso a todas las cuentas de usuario del sistema, lo que no es razonable salvo para el superusuario (o *root*, o *uid* `0`).

La respuesta suele ser que el fichero con el ejecutable de *passwd* tiene un permiso puesto, el *set uid bit*, que indica que dicho comando debe ejecutar de forma efectiva a nombre del dueño del fichero, y no a nombre del usuario que ejecuta dicho fichero. Dado que `/bin/passwd` (o el fichero de que se trate) pertenece normalmente a *root*, cualquier usuario puede cambiar su password.

Fíjate en esto:

```
unix$ which passwd
/usr/bin/passwd
unix$ ls -l /usr/bin/passwd
-r-sr-xr-x  1 root  bin  23800 Mar  8  2015 /usr/bin/passwd*
```

El comando *which(1)* imprime el nombre del fichero que corresponde a un comando dado. Como puedes

ver, hay una `s` como permiso de ejecución para el dueño de `passwd`. Eso quiere decir que cualquier usuario que ejecute dicho fichero obtiene un proceso a nombre de *root*.

Podemos poner o quitar ese permiso como cualquier otro, siempre que tengamos permiso para ello:

```
unix$ cc -o eco eco.c
unix$ chmod u+s eco
```

Existe otro bit similar para el grupo, se llama el *set gid bit*, y se puede activar como en...

```
unix$ chmod g+s eco
```

Ahora podemos decir que en realidad los procesos tienen dos *uid* y dos *gid*. Tienen los reales y tienen los *efectivos*. Las llamadas *setuid(2)* y *seteuid(2)* permiten cambiar los *uid* real y efectivo de un proceso, y las llamadas *setgid(2)* y *setegid(2)* permiten cambiar los *gid* real y efectivo.

Como podrás suponer, en realidad son los identificadores efectivos los que se utilizan para comprobar permisos. Los reales suelen utilizarse para saber quién está en realidad ejecutando qué.

El comando *su(1)* (o *switch user*) permite ejecutar un shell a nombre de otro usuario (de *root* si no se indica nombre de usuario) y naturalmente utiliza el mecanismo de *set uid* para conseguirlo. Por ejemplo:

```
unix$ su
Password:
unix# id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon)
unix# exit
unix$
```

3. ¿Qué más tiene un proceso?

En este punto sabes que los procesos tienen un *pid*, segmentos de memoria, valores para los registros (mientras no ejecutan, ¡cuando ejecutan dichos valores se guardan en los registros de verdad!), un directorio actual, variables de entorno, el identificador de usuario o *uid* a nombre del cual ejecutan, etc.

De aquí en adelante veremos otros recursos que tienen los procesos tales como ficheros abiertos y otros muchos. Piensa siempre que se trata de otros campos en el record que implementa cada proceso e intenta imaginar las estructuras de datos que los implementan. Es una buena forma de que vea que no hay magia por ningún lado.

Pero... ¿Cómo vas a ser capaz de recordar todo esto? ¡No lo hagas! ¡Usa el manual!