Práctica 4. Llamadas al sistema

J.C. Pérez, S. Sáez, V. Lorente y L. Pascual

Índice

1. Introducción

El objetivo de esta práctica es añadir una nueva funcionalidad al núcleo de Linux mediante la implementación de una nueva llamada al sistema. Hay que abordar necesariamente esta implementación desde dos frentes.

1. Añadir la llamada al sistema en el núcleo del sistema operativo.

   Con esto conseguiremos que, cuando un proceso cargue cierto valor en el registro eax y lance a continuación la interrupción software **0x80**, la rutina de servicio de esta interrupción ejecute la función del núcleo que queramos (normalmente una función escrita ex-profeso).

2. Crear la interfaz C de la llamada al sistema.

   Y con esto lograremos que cualquier programa pueda realizar la llamada al sistema mediante la invocación de un función normal de C con sus parámetros, etc.

En esta práctica no se crean módulos, sino que se modifican directamente ficheros fuente del núcleo. Por ello, es importane seguir estas normas:

- Cada grupo borrará siempre antes de irse el directorio de los fuentes de linux:

```
# rm -rf /usr/src/linux
```

- Si al llegar existe este directorio, se borrará inmediatamente.
- El fichero **/usr/src/linux-version.tar.gz** no se borrará ni modificará. De ese modo no hace falta bajarlo cada sesión.
- La primera operación, por tanto, antes de empezar con las modificaciones, será siempre desempaquetar el núcleo:

```
# cd /usr/src
# tar xzvf linux-version.tar.gz
```

Recordemos que en las sesiones de prácticas trabajamos con la versión de núcleo 2.6.20 y que en clase explicamos la 2.6.11, por tanto, debemos observar que hay ciertos ficheros que han cambiado un poco, aunque el modo en que funcionan las llamadas al sistema no lo ha hecho.

2. Funcionalidad de la llamada al sistema

La nueva llamada al sistema debe devolver la generación que ocupa un proceso dentro del árbol genealógico de procesos, es decir, el número de ancestros que hay que atravesar en el árbol de procesos hasta alcanzar la raíz: el proceso **init** (`pid == 1`).

Veamos un ejemplo analizando la salida del comando `ps 1`:

```
...   PID  PPID PRI  ... TIME COMMAND
...   150     1   0  ... 0:00 bash
...   556   150   0  ... 0:00 xinit /root/.xinitrc
...   564   556   1  ... 0:01 kwm
...   573   564   0  ... 0:00 kaudioserver
...   579   564   0  ... 0:04 kpanel
...   581   573   0  ... 0:00 maudio -media 129
```

PID identifica al proceso y PPID al padre de ese proceso. En este caso, si la llamada la invocara el proceso `bash`, con PID 150, el resultado debería ser 1, pues es hijo directo del proceso **init** (PID 1). La siguiente tabla muestra el resultado que se esperaría en el resto de los procesos:

PID	PPID	COMMAND	Resultado
556	150	xinit /root/.xinitrc	2
564	556	kwm	3
573	564	kaudioserver	4
579	564	kpanel	4
581	573	maudio -media 129	5

Una posible página para esta llamada en el manual en línea de UNIX sería:

```
generacion(2)     Linux Programmer's Manual         generacion(2)

NAME
      generación - obtiene la generación a la que  pertenece  el
               proceso invocante

SYNOPSIS

      #include <unistd.h>

      int generacion(void);

DESCRIPTION

      generacion devuelve  la   profundidad   dentro  del  árbol
      genealógico de procesos, es decir, el  número de  procesos
      que  hay  que  atravesar  desde  el  proceso  init   hasta
      alcanzar el proceso invocante.

CONFORMING TO

      ESO/DSO

SEE ALSO
      exec(3), fork(2), getpid(2), getppid(2)

Linux                      December 8, 2007
```

3. Implementación: nivel de núcleo del SO

Una vez detallada la funcionalidad que se espera de la nueva llamada al sistema, vamos entrar en los detalles de implementación. Primero veremos cómo construir la función en C, que implementaremos dentro del

núcleo, y que será la que haga todo el trabajo, y después veremos cómo dar de alta dicha función dentro del núcleo para ofrecerla como llamada al sistema.

Todos los procesos del sistema tienen asociada, dentro del núcleo, una estructura que contiene toda la información relevante de dicho proceso (struct task_struct). Durante esta práctica denominaremos a dicha estructura descriptor de proceso. Se suele trabajar con punteros a la tabla de procesos, luego manejaremos a menudo variables de tipo puntero a descriptor de proceso (**`struct task_struct *`**). La macro current contiene el puntero correspondiente al proceso que está actualmente en ejecución.

3.1. El árbol genealógico de procesos

El descriptor de un proceso (**`struct task_struct`**) contiene información sobre el proceso al que representa: su **`pid`**, el identificador de usuario, uid, el identificador de grupo, gid, los descriptores de ficheros abiertos, y un largo etc.. Entre toda la información que contiene cabe destacar los siguientes campos:

```
862     /*
863      * pointers to (original) parent process, youngest child, younger sibling,
864      * older sibling, respectively. (p->father can be replaced with
865      * p->parent->pid)
866      */
867      struct task_struct *real_parent; /* real parent process (when being debugged) */
868      struct task_struct *parent; /* parent process */
869     /*
870      * children/sibling forms the list of my children plus the
871      * tasks I'm ptracing.
872      */
873      struct list_head children; /* list of my children */
874      struct list_head sibling; /* linkage in my parent's children list */
875      struct task_struct *group_leader; /* threadgroup leader */
```

Estos campos permiten organizar los procesos en un árbol genealógico. Con el comando **`pstree`** podemos obtener un árbol similar que muestra únicamente la relación de parentesco:

```
init-+-apache---6*[apache]
     |-apmd
     |-atd
     |-bash---xinit-+-XF86_SVGA
     |              `-kwm-+-kaudioserver---maudio
     |                    |-kbgndwm
     |                    |-kfm---kvt---bash-+-pstree
     |                    |                  `-xemacs---ispell
     |                    |-kpanel
     |                    |-krootwm
     |                    `-kwmsound
     |-cron
     |-5*[getty]
     |-gpm
     |-inetd
     |-kflushd
     |-klogd
     |-kswapd
     |-lpd
     |-portmap
     |-sshd
     |-syslogd
     `-update
```

Con **`pstree -p`** obtendríamos también los PIDs de los procesos del árbol. Haz **`man pstree`** para ver más opciones.

Por ejemplo, en este caso, el proceso **`XF86_SVGA`** tendría en la entrada **`parent`** de su descriptor de proceso un puntero al descriptor del proceso **`xinit`**. Utilizando el campo parent se puede recorrer el árbol genealógico

de procesos en sentido ascendente, y así, podemos contar los saltos que existen hasta llegar al proceso **init**, que es el objetivo de la nueva llamada al sistema.

3.2. Implementación de la función interna `sys_generacion`

La función que implementará la nueva llamada al sistema **`sys_generacion`** se puede añadir, por ejemplo, al fichero **`kernel/sys.c`**, que contiene la implementación de algunas de las llamadas al sistema.

La llamada al sistema tendrá una estructura similar a la siguiente:

```
asmlinkage int sys_generacion(void)
{
  int generacion;

  ...

  /* return generacion; */

} /* end sys_generacion */
```

3.3. Dar de alta la llamada al sistema en el núcleo

Para dar de alta la llamada en el núcleo de Linux deberemos añadir en el fichero **`arch/x86/include/asm/unistd_32.h`** el identificador de la llamada y en el fichero **`arch/x86/kernel/syscall_table_32.S`** el puntero a la función sys_generacion que implementa la llamada al sistema. Para saber con seguridad cómo y dónde insertar este puntero debes entender qué es **`sys_call_table`** y cómo se usa en el núcleo (examina la rutina de servicio de la interrupción software **`0x80`**, **`ENTRY(system_call)`** en el fichero **`arch/x86/kernel/entry_32.S`**).

En la tabla del fichero **`arch/x86/kernel/syscall_table_32.S`** se rellenan algunas entradas reservadas y otras tienen punteros a la función **`sys_ni_syscall`** que no hace nada excepto devolver un error de "llamada inexistente" (**`-ENOSYS`**). Intenta entender dónde y cómo se rellena el resto de la tabla.

además de definir la macro de nuestra llamada al sistema:

```
#define __NR_generacion num_llamada
```

¿Al añadir nuestra llamada al sistema, es necesario actualizar el número NR_syscalls?

Una vez implementada la función, y dada de alta en el núcleo del sistema, sólo queda recompilar el núcleo de Linux, como se hizo en la primera práctica, y rearrancar.

4. Implementación: nivel usuario

Para poder utilizar la nueva llamada al sistema desde un programa en C disponemos de la función **`int syscall(int number, ...)`**.

En la entrada del manual **`man syscall`** vemos que basta con llamar a esta función con el número o, preferiblemente, el nombre que se ha definido en **`unistd.h`**.

También vemos que necesitaremos incluir, además de **`<unistd.h>`**, **`<sys/syscall.h>`**

Para comprobar la corrección de la nueva llamada al sistema, se realizará un programa de prueba que invoque nuestra llamada a través de esa interfaz en C. El programa deberá devolver a qué generación pertenece él mismo.

```
# mi_generacion
Soy de la generación 5.
```

5. Llamada al sistema generacion

El objetivo final de la práctica consiste en modificar la sencilla llamada al sistema anterior de forma que reciba un **pid** como parámetro y devuelva el número de generación del proceso con ese **pid**.

Para recorrer la lista de procesos se puede utilizar la macro **for_each_process**, que está en **include/linux/sched.h**. ¡No dejéis de examinar y entender la definición de esta macro!

Veamos un ejemplo de uso:

```
struct task_struct *p;
...
for_each_process(p) {

  if (p->pid == ALGO)
    {
      /* HACER LO QUE SEA NECESARIO */

      break;
    } /* endif */

} /* end for_each_process */
```

En caso de que no exista un proceso con el **pid** indicado, la función deberá devolver un error del tipo **-ESRCH**, que le indicará al usuario que no existe dicho proceso.

Al igual que antes, se deberá construir una nueva orden generacion, que admita como parámetro (No se os ocurra pedir interactivamente el **pid**. ¡Esto es UNIX, no un sistema "de juguete"!. Podéis consultar el paso de argumentos en un manual de C. Por ejemplo aquí. el **pid** de un proceso, y que devuelva un mensaje indicando, o bien la generación de dicho proceso, o bien el mensaje de error correspondiente, indicando el valor de la variable errno.

Utilizad **pstree -p** para ver qué procesos tenéis en marcha y probad la orden con varios de ellos.

Probad también con números de proceso inexistentes y con el proceso **init**, para comprobar que habéis tenido en cuenta adecuadamente estos casos particulares.