http://lsi.ugr.es

Diseño de Sistemas Operativos
5º Curso de Ingeniería Informática

Guión de Prácticas - Curso 2010-11

Práctica 3:

Implementación de una llamada al sistema en Linux

Objetivo de la práctica:

Conocer como se puede añadir nueva funcionalidad al kernel de sistema operativo y cómo obtener acceso a ella mediante una nueva llamada al sistema.

1 Introducción

Añadir una nueva llamada al sistema consta de dos pasos (que detallaremos en el Apartado 3):

a) **Añadir la nueva llamada al núcleo del sistema operativo** – es decir, que cuando se invoque a la nueva llamada invocando a la instrucción `int 0x80` después de haber cargado en el registro `eax` el número que identifica a la llamada, el kernel ejecuta la función que hemos escrito para suministrar la nueva funcionalidad.

b) **Crear una interfaz en C para la nueva llamada** – esto permite a los programas en C hacer uso de la nueva llamada sin necesidad de tener que acceder a la llamada a través del lenguaje ensamblador.

2 Trabajo propuesto en la práctica

La práctica propuesta pretende implementar una nueva llamada que nos devuelva el número de generación del proceso que realiza la llamada al sistema dentro del árbol de procesos del sistema hasta alcanzar el proceso raíz, el *init* cuyo PID es 1.

Podemos observar el árbol de procesos mediante la orden `pstree`. La llamada al sistema nos devolverá el nivel en el que se encuentra el proceso que la invoca. Por ejemplo, la ejecución de `pstree -p` en mi sistema se muestra en la figura. Podemos ver como el proceso `more` tiene como número de generación el 3.

```
[jagomez@localhost jagomez]$ pstree -p |more
init(1)-+-apmd(713)
        |-atd(774)
        |-automount(762)
        |-bdflush(6)
        |-crond(849)
        |-gpm(837)
        |-kapm-idled(3)
        |-keventd(2)
        |-khubd(78)
        |-klogd(600)
        |-kreclaimd(5)
        |-kswapd(4)
        |-kupdated(7)
        |-login(910)---bash(922)-+-more(964)
        |                        `-pstree(963)
        |-mdrecoveryd(8)
        |-mingetty(911)
        |-mingetty(912)
        |-mingetty(913)
        |-mingetty(914)
        |-mingetty(915)
        |-portmap(614)
        |-pump(531)
--Más--
```

3 Material necesario

Toda la información de un relevante para la realización de la práctica se encuentra en la estructura `task_struct` , que corresponde al *bloque de control del proceso* o ***descriptor***. Para el proceso actual en ejecución la macro `current` (en *include/linux/sched.h*) determina su descriptor. Dicho descriptor tiene múltiples campos de los cuales solo necesitaremos:

`pid`	– el PID del proceso
`p_opptr`	– el proceso padre original (`parent` en Red Hat)
`p_pptr`	– el proceso padre (`real_parent` en Red Hat)
`p_cptr`	– el proceso hijo más joven (`children` en Red Hat)
`p_ysptr`	– el proceso hermano inmediatamente más joven (`sibling` en Red Hat)
`p_osptr`	– el proceso hermano inmediatamente más viejo

Esto campos permiten trazar el árbol genealógico de un proceso hasta el origen común, el *init*. En concreto `p_pptr` nos permite recorrer el árbol en sentido ascendente, y por tanto podemos contar los saltos hasta el *init*.

A continuación veremos con detalle los pasos a seguir para implementar la llamada tanto a nivel kernel como a nivel de interfaz de biblioteca.

A) Implementación a nivel de núcleo

- Implementación de la función interna

Las funciones correspondientes a llamadas al sistema siempre empiezan por `sys_`. En nuestro caso, la función se llamará `sys_generacion`. Nuestra función se puede añadir en el archivo *kernel/sys.c* donde esta el código de otras llamadas al sistema y la añadiremos al final.

```
asmlinkage int sys_generation(void)
{
      …
      int generacion;
      …
      return generacion;
}
```

- Dar de alta la llamada al sistema en el núcleo

Para dar de alta la nueva función en el kernel debemos:

1. Añadir el identificador de la llamada en el archivo *include/asm-i386/unistd.h*.
2. Añadir en el archivo *arch/i386/kernel/entry.S* el puntero a la función `sys_generacion` que implementa nuestra llamada al sistema. Para saber cómo y donde insertar este puntero debemos entender que es `sys_call_table` (en *kernel/ksyms.c* como ya vimos en la práctica anterior) y cómo se utiliza en el núcleo (podeis examinar la rutina de servicio de la interrupción software 0x80 -`ENTRY(system_call)` en *arch/i386/kernel/entry.S*.
3. Actualizar el número de llamadas al sistema implementadas en *arch/i386/kernel/entry.S* al final del archivo, para que se rellene el resto de la tabla con entradas nulas.

Determinar si es necesario o no modificar el valor actual de `NR_syscalls` en *include/linux/sys.h*.

- Compilar el kernel

Una vez completados los pasos anteriores, solo nos queda recompilar el núcleo e re-instalarlo como hemos realizado en ocasiones anteriores. Recordar que es muy importante revisar bien el

código de la llamada al sistema, puesto que se ejecutará en modo kernel y si esta falla por alguna razón o queda en un bucle infinito, el sistema no se recuperará (el kernel que utilizamos es no apropiativo por lo que si falla el sistema podrá detenerse y tendremos que rearrancar).

B) Implementación a nivel de biblioteca

La creación de la interfaz de la llamada al sistema es sencilla. El mecanismo viene descrito con detalle, incluido un ejemplo, en la Sección 2 del manual en línea y que debemos consultar con la orden `man 2 intro`.

En teoría, la interfaz de la llamada debería ser incluida en la biblioteca *libc*, pero en nuestro caso no lo haremos. Lo que haremos será incorporar la interfaz en el mismo programa que usará la llamada o bien en un archivo, por ejemplo, con nombre *generacion.c*, que generará un objeto que enlazaremos posteriormente con nuestro programa.

- Programa de prueba

Para comprobar el funcionamiento de nuestra llamada, se creará un programa de prueba que imprimirá en pantalla la generación del propio programa. Por ejemplo:

```
% ./mi_generacion
Soy de la generación 5.
```

- Ampliación de la funcionalidad de la llamada al sistema (realización opcional)

Una ve que funcionen todos los elementos anteriores, dotaremos a la llamada de una mayor funcionalidad. La ampliaremos para que además de la generación del programa llamador nos de la generación de cualquier programa cuyo PID pasamos como argumento. Para su realización será necesario modificar tanto el código de la llamada como la interfaz C de la función.

Para recorrer la lista de procesos se utilizará la macro `for_each_task` definida en el archivo *include/linux/sched.h*. Un ejemplo de uso:

```
struct task_struct *p;
…
for_each_task(p)  {
      if (pid_PID == ALGO) {
            /* hacer lo que sea necesario */
            break;
      }  /* fin del if */
} /* fin del for_each_task */
```

En caso de que no exista un proceso con el PID dado como argumento, la función devolverá el error `-ESRCH` (*include/asm-i386/errno.h*) indicando que no existe tal proceso.

Debemos construir de nuevo la función generación para que admita un parámetro el PID (no debemos pedir esta valor de forma interactiva) y que devuelva el valor de generación o el código de error correspondiente en la variable `errno`.

Para probar la llamada implementada podemos utilizar la orden `pstree -p` para ver los procesos de nuestro sistema y probar con varios de ellos. Probar también identificadores de procesos inexistentes, y con el *init*, para comprobar que habéis controlado correctamente los casos particulares.