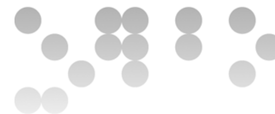


PRÁCTICA 2: MODIFICACIONES AL KERNEL

Presentación

Las prácticas de las asignaturas Ampliación de Sistemas Operativos y Diseño de Sistemas Operativos pretenden que el estudiante sea capaz de entender porciones de cógido fuente de una versión actual del *kernel* de Linux y de introducir modificaciones localizadas en las funcionalidades del *kernel*.

Las prácticas de la asignatura son dos:

- La primera práctica describe cómo preparar el entorno de desarrollo utilizado en las prácticas (compilación del *kernel* y arranque de una máquina utilizando el *kernel* generado) y cómo navegar en la estructura de directorios que componen el código fuente del *kernel* de Linux. También presenta el mecanismo de los módulos porque será utilizado en la segunda parte de la práctica para introducir nuevas funcionalidades al *kernel*.
- La segunda práctica propone realizar una serie de modificaciones localizadas en el código del *kernel* de Linux. Por ejemplo, recorrer la estructura de tablas de páginas de un proceso, crear un driver para un nuevo dispositivo u obtener información sobre el sistema de ficheros.

Los conocimientos previos necesarios para el desarrollo de estas prácticas son:

- Algorítmica.
- Programación en un lenguaje de alto nivel (preferentemente lenguaje C). Uso de punteros.
- Estructuras de datos (listas doblemente encadenadas, tablas hash).
- Conceptos teóricos de la asignatura Sistemas Operativos.
- Utilización de Linux desde el intérprete de órdenes (`shell`).
- Entender pequeños fragmentos de código escritos en assembler i386.
- Los presentados en la práctica 1.

Este documento contiene el enunciado de la segunda práctica.

La segunda práctica de la asignatura propone que el estudiante escriba código que pase a formar parte del *kernel* de Linux. Para realizar esta tarea se utilizará el entorno de desarrollo presentado en la práctica anterior (máquina *host* y máquina *guest*) y el mecanismo de los módulos. Para hacer esta práctica es preciso descargar el fichero `base2.zip` de [1] y descompactarlo con la orden `unzip`.

Competencias

Transversales:

- Capacidad para adaptarse a las tecnologías y a los futuros entornos actualizando las competencias profesionales
- Capacidad de comunicación escrita en el ámbito académico y profesional

Específicas:

- Capacidad de analizar un problema en el nivel de abstracción adecuado a cada situación y aplicar las habilidades y conocimientos adquiridos para abordarlo y resolverlo
- Capacidad de diseñar y construir aplicaciones informáticas mediante técnicas de desarrollo, integración y reutilización

Objetivos

Los objetivos de esta práctica son que el estudiante...

- conozca los pasos a seguir para que el *kernel* pueda ofrecer una nueva llamada al sistema.
- implemente el *driver* para un nuevo dispositivo.
- explore el código fuente del *kernel* para estudiar fragmentos similares a los que deberá implementar.

Para que el profesor pueda evaluar el grado de cumplimiento de estos objetivos, los estudiantes deberán resolver *individualmente* dos ejercicios.

Descripción de la Práctica a realizar

1. Ejercicio 1: Añadir una nueva llamada al sistema

Linux 3.16.2 consta, aproximadamente, de 330 llamadas al sistema. Cada llamada tiene asociado un identificador numérico (pueden consultarse en el fichero `arch/x86/include/generated/asm/syscalls_32.h`).

Para añadir una nueva llamada al sistema al *kernel* se puede proceder de dos formas: recompilando el *kernel* o creando un módulo. Aunque la primera forma sería la solución más genérica, en esta práctica se propone utilizar la segunda forma. Se aprovechará el hecho que, realmente, algunos de los identificadores de llamadas al sistema no están siendo utilizados.

Antes de comenzar el ejercicio es preciso compilar el *kernel* para permitir que los módulos puedan utilizar los símbolos del *kernel* `sys_call_table` y `sys_ni_syscall`. Los pasos a seguir son:

- Copiar el fichero `i386_ksyms_32.c` (contenido en `base2.zip`) al directorio del código fuente del *kernel* `arch/x86/kernel`.
- Desde el directorio `linux-3.16.2`, compilar el *kernel* ejecutando la orden `make`.

1.1. Ejemplo 4: Llamada al sistema vacía

Os facilitamos el código fuente de un módulo (`modules/example4/newsyscall.c`) que añade una nueva llamada al sistema. El módulo, al ser instalado, realiza las siguientes tareas:

1. Busca una entrada no utilizada en la tabla `sys_call_table`. Para encontrarla hay que recorrer la tabla `sys_call_table` y buscar alguna entrada que esté inicializada con el valor `sys_ni_syscall`). Las entradas de esta tabla son de tipo `unsigned` y la tabla tiene `NR_syscalls` entradas.
2. Muestra (utilizando la rutina `printk`) el índice (número de posición) correspondiente a la entrada encontrada. Este índice será el identificador que se asociará a la nueva llamada al sistema. En condiciones normales, este índice será el 17.
3. Substituye el contenido de la entrada no utilizada de `sys_call_table` por la dirección de memoria de una rutina (`sys_newsyscall`, implementada en el módulo) que será la que dé servicio a la nueva llamada al sistema. En este ejemplo, la rutina imprime `Hello world` seguido de un parámetro.

Al desinstalar el módulo, se restaurará el contenido de la entrada utilizada de la tabla `sys_call_table`.

Para conseguir que un programa de usuario pueda invocar a la nueva llamada al sistema hay que utilizar la llamada al sistema indirecta `syscall`. Está parametrizada con el identificador numérico de la llamada al sistema a invocar y con la lista de parámetros de la llamada. Por ejemplo, podemos invocar `read(fd, &c, 1)` mediante `syscall(__NR_read, fd, &c, 1)`. El programa de usuario `modules/example4/test4.c` invoca a la nueva llamada al sistema (asumiendo que está instalada en la posición 17).

La orden `make` en la máquina *host* compila el módulo y el programa de pruebas. Los ficheros `newsyscall.ko` y `test4` deben ser copiados a la máquina *guest*. Tras instalar el módulo, la ejecución de `test4` provocará la aparición del mensaje `Hello world` mostrado por la rutina que da servicio a la nueva llamada al sistema. Además, también mostrará el parámetro especificado en la línea de órdenes.

1.2. Llamada al sistema a implementar

Una vez entendido el ejemplo anterior, se pide que lo modifiquéis para que la nueva llamada al sistema implemente un servicio más útil.

El servicio a implementar será informar sobre el número de canales (*file descriptors*, descriptores de ficheros) que tiene abiertos un determinado proceso. La llamada tendrá la interfície `int newsyscall(pid_t pid)` donde `pid` es el identificador de proceso del que volemos obtener esta información. Si el proceso no existe, la llamada debe retornar el código de error `-ESRCH`; en otro caso, la llamada devolverá el número de canales abiertos que tiene el proceso.

Para calcular esta información os proporcionamos la rutina `int num_open_files(struct task_struct *task)` que, dado el `struct task_struct *` (puntero al Process Control Block) de un proceso, nos devuelve el número de canales que tiene abiertos. También debéis utilizar la rutina del *kernel* `struct task_struct *find_task_by_vpid(pid_t vnr)` que, dado un identificador de proceso, retorna el puntero a su `struct task_struct` o `NULL` si no existe.

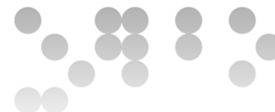
Utilizando `cscope` (u otras herramientas) es posible ver la definición y ejemplos de uso de `find_task_by_vpid`.

El programa de usuario `modules/newsyscall/testnew.c` invoca a la nueva llamada y realiza algunas pruebas para determinar su correcto funcionamiento. Si la ejecución del programa no llega hasta su última sentencia, indica que vuestro código contiene algún error.

1.3. Entregas correspondientes a este ejercicio

Las entregas correspondientes a este ejercicio son:

- El código fuente del módulo que implementa la nueva llamada al sistema.



- Breve explicación del trabajo desarrollado (formato `pdf`, incluyendo captura de pantalla del resultado del programa de pruebas).

2. Ejercicio 2: Implementación de un *driver*

El módulo didáctico *La gestión de las entradas/salidas* de la documentación de la asignatura ([2]) presenta el concepto de *driver* (capítulo 1) y el de descriptor de dispositivo (capítulo 5.3.2). Los *drivers* de Linux utilizan una estructura de datos similar a un descriptor de dispositivo. Por tanto, la implementación de las llamadas al sistema de entrada/salida está estructurada en dos niveles: uno independiente del dispositivo y otro dependiente del dispositivo.

El apartado 2.1 presenta el ejemplo de un módulo que da de alta un nuevo *driver*. El apartado 2.2 plantea el ejercicio a resolver en esta práctica.

2.1. Ejemplo 5: *driver*

En el directorio `modules/example5` tenéis el código fuente de un módulo (`abc.c`) que da de alta dinámicamente un nuevo *driver* en Linux. Este *driver* retorna ordenadamente las letras del alfabeto. Para probar el módulo, disponéis del programa (`test5.c`) que pone a prueba las llamadas al sistema que implementa el *driver*. La ordern `make` compila tanto el módulo como el programa de usuario. Una vez compilados, deben ser transferidos a la máquina *guest* para ser probados.

A continuación se describirá el código fuente del módulo. La rutina de instalación del módulo indica al sistema operativo que se quiere registrar un nuevo *driver*. Ésto se realiza con la rutina `register_chrdev()`. Esta rutina está parametrizada con el *major* del dispositivo, el nombre del dispositivo y con un puntero a una estructura del tipo `struct file_operations`.

- El *major* es un número entero que identifica al dispositivo. Los *majors* utilizados están definidos en el fichero `include/linux/major.h`. Es preciso escoger uno libre y definir el valor de la macro `DRIVER_MAJOR` en el fichero `abc.c`. Escogemos el 231.
- El nombre del dispositivo no es más que un *string* que se utilizará cuando se listen los dispositivos activos (fichero `/proc/devices`).
- El puntero a una estructura del tipo `struct file_operations` permitirá que el *kernel* conozca cuáles son las rutinas específicas para acceder al dispositivo. La estructura tiene punteros a las operaciones `open()`, `read()`, `write()`, `llseek()`, `ioctl()`,... específicas del dispositivo; todas estas rutinas tienen una interfície definida. Antes de invocar a la rutina `register_chrdev()`, es necesario inicializar dicha estructura.

Una vez instalado el módulo (y registrado el *driver*) en la máquina *guest*, hay que crear un fichero de tipo dispositivo que esté asociado al *major* indicado como primer parámetro en la rutina `register_chrdev()`. Una posible forma de hacerlo es utilizando la orden `mknod /dev/ex5 c 231 0` en la máquina *guest*. A partir de este momento, todas las operaciones de entrada/salida (llamadas al sistema `open()`, `read()`, `write()`,...) hechas sobre este fichero serán servidas por las rutinas especificadas en el tercer parámetro de la rutina `register_chrdev()`.

La rutina `open()` del dispositivo comprueba que el fichero haya sido abierto en modo `O_RDONLY` (campo `f_flags` del segundo parámetro de la rutina) y retorna 0, en otro caso retorna el resultado `-EACCES` (resultados negativos indican que se ha producido algún error y, en valor absoluto, cuál es el código de error, resultados mayores o iguales que cero indican que la llamada se ha realizado correctamente),

La rutina `read()` del dispositivo tiene cuatro parámetros: un puntero a una estructura `struct file`, la dirección de memoria del buffer de usuario donde se han de escribir los caracteres leídos, el número máximo de caracteres a leer y la dirección de memoria donde se almacena el puntero de lectura/escritura sobre el fichero. El segundo y el tercer parámetro de la llamada `read()` del dispositivo coinciden con el segundo y tercer parámetro de la llamada al sistema `read()` invocada por el programa de usuario. En función del valor del puntero de lectura/escritura sobre el fichero (parámetro `f_pos`), escribe sobre el buffer de usuario los caracteres correspondientes mediante la rutina `copy_to_user`. Finalmente, la rutina retorna el número de caracteres que han sido leídos. Además, el *driver* limita a tres el número máximo de caracteres que pueden ser leídos en cada lectura.

La rutina `llseek()` del dispositivo actualiza el puntero de lectura/escritura en función de los parámetros `offset` y `orig`. La rutina retorna el nuevo valor del puntero de lectura/escritura (o `-EINVAL` si hay error).

Al desinstalar el módulo hay que desregistrar el *driver* utilizando la rutina `unregister_chrdev()`.

2.2. *driver* a implementar

Una vez hayáis entendido el código de ejemplo del *driver* presentado en la sección anterior, podéis resolver el problema siguiente.

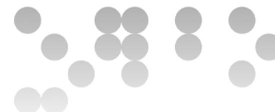
Se pide que transforméis el módulo del ejercicio anterior (el que añadía una nueva llamada al sistema que retornaba el número de canales que tenía abiertos un proceso) de forma que **no** interaccionéis con él utilizando la nueva llamada al sistema sino que lo hagáis utilizando el fichero dispositivo `/dev/openfiles`). Para hacerlo, podéis aprovechar gran parte del *driver* de ejemplo `abc.c`.

El *driver* deberá responder a las siguientes llamadas al sistema:

- `open()`: Para poder acceder a la información será preciso abrir el dispositivo utilizando la llamada al sistema `open` en modo `O_RDONLY`. Si se intenta abrir en cualquier otro modo que no sea `O_RDONLY`, la llamada retornará el error `EACCES`. En caso que el dispositivo ya esté abierto, la llamada retornará el error `EBUSY`.
- `read()`: Para leer la información será necesario utilizar la llamada al sistema `read(int fd, char *buf, int num)`.

  Se interpretará el puntero de lectura/escritura asociado al fichero dispositivo como el *pid* del proceso del que queremos obtener su número de canales abiertos. El tercer parámetro de `read` se interpretará como el número máximo de procesos de los que queremos obtener la información (no de bytes porque estamos en una arquitectura donde los enteros ocupan 4 bytes). Por ejemplo, si el puntero de lectura/escritura apunta al proceso 54 y el tercer parámetro de `read` vale 3, la llamada intentará leer los contadores de los procesos 54, 55 y 56, es decir, 12 bytes. Puede pasar que no podamos leer los contadores de todos los procesos porque algún proceso ya no exista. El segundo parámetro de `read` se interpretará como la dirección de memoria del espacio de memoria de usuario a partir de la cual se dejarán los contadores leídos (-1 en caso que el proceso no exista). Como en la arquitectura utilizada los datos de tipo integer son de cuatro bytes, el número de bytes a leer deberá ser múltiplo de cuatro; en caso contrario, la llamada al sistema retornará el error `EINVAL`. La llamada retornará como resultado el número de contadores que han sido leídos.

  Un efecto lateral de esta llamada será incrementar el puntero de lectura/escritura sobre el fichero.
- `lseek()`: La llamada `lseek` modificará el puntero de lectura sobre el dispositivo de forma similar a como permite modificarlo sobre un fichero ordinario. De esta forma, podremos posicionarnos para leer los datos relativos a un proceso concreto. Se admitirán los desplazamientos `SEEK_CUR` y `SEEK_SET`, pero no `SEEK_END`.
- `close()`: La llamada `close` liberará el dispositivo para que pueda ser abierto por otros procesos.



Observaciones:

- Como el código del *driver* forma parte del *kernel*, es importantísimo realizar un control de errores exhaustivo y retornar el código de error adecuado a cada caso.
- Podéis utilizar la rutina `printk` para imprimir chivatos en el código del módulo.
- Disponéis de un programa de prueba (`modules/driver/testdriver.c`) que comprueba el funcionamiento del *driver*. Para generar el ejecutable hay que ejecutar `make`. Si el programa de prueba aborta antes de ejecutar la última sentencia, vuestro *driver* no es correcto.
- En caso de duda respecto al comportamiento de las llamadas al sistema o de los códigos de error que han de retornar, consultad el código del juego de pruebas.

2.3. Entregas correspondientes a este ejercicio

Las entregas correspondientes a este ejercicio son:

- El código fuente del módulo que implementa el *driver*.
- Breve explicación del trabajo desarrollado (formato `pdf`, incluyendo captura de pantalla del resultado del programa de pruebas).

Recursos

Recursos Básicos

[1] Ficheros de ejemplo y shellscripts para la segunda práctica.
URL `http://einfmlinux1.uoc.edu/aso/base2.zip`

Recursos Complementarios

[2] T. Jové, J. L. Marco, D. Royo, E. Morancho, Ampliación de Sistemas Operativos.

Criterios de valoración

El primer ejercicio tiene un peso del 40 % y el segundo del 60 %.

Formato y fecha de entrega

Las respuestas a esta práctica se entregarán en un único fichero `zip` o `tar` resultado de comprimir dos directorios. Cada directorio corresponderá a un ejercicio y contendrá los ficheros a entregar del ejercicio. Este fichero debe entregarse en el registro de evaluación continua antes de la fecha límite establecida en el plan docente (24:00 del 23 de diciembre).