PRÁCTICA 2: MODIFICACIONES AL KERNEL

Presentación

Las prácticas de la asignatura Diseño de Sistemas Operativos pretenden que el estudiante sea capaz de entender porciones de código fuente de una versión actual del kernel de Linux y de introducir modificaciones localizadas a 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 el arranque de una máquina utilizando el kernel generado) y como navegar dentro de la estructura de directorios que componen el código fuente del kernel de Linux. También presenta el mecanismo de los módulos perqué será utilizado en la segunda práctica para introducir nuevas funcionalidades al código del kernel.

• La segunda práctica propone realizar una serie de modificaciones bastante localizadas en el código del kernel de Linux. Por ejemplo, recorrer la estructura de tablas de páginas de un proceso, crear un driver para a un nuevo dispositivo u obtener información sobre el sistema de archivos.

Los conocimientos previos necesarios para el desarrollo de estas prácticas son:
• Algorítmica.
• Programación en un lenguaje de alto nivel (preferentemente en 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 pedidos.
• Entender pequeños fragmentos de código escritos en ensamblador i386.
• Los presentados a 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 utilitzará el entorno de desarrollo presentado en la práctica anterior (máquina host y maquina guest) y el mecanismo de los módulos. Para hacer esta práctica hay que descargar el archivo base2.zip de [1] Y des compactarlo con el pedido unzip.

Competencias

Transversales:

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

Especıficas:

• Capacidad para analizar un problema en el nivel de abstracción adecuado a cada situación y aplicar las habilidades y conocimientos adquiridos para abordarlo y resolverlo
• Capacidad para 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 porqué 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 de código similares a los que deberá de implementar.

Porqué el profesor pueda evaluar el grado de consecución de estos objetivos, los estudiantes deberán resolver individualmente dos ejercicios.

Descripción de la Práctica

1 Ejercicio 1: Añadir una nueva llamada al sistema

Linux 4.7.5 consta, aproximadamente, de 380 llamadas al sistema. Todas las llamadas tienen asociado un identificador numérico (pueden consultarse en el archivo arch / x86 / include / generated / asm / syscall 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 de que, realmente, algunos de los identificadores de llamadas al sistema no están utilizados.

Antes de comenzar el ejercicio hay que compilar el kernel para permitir que los módulos puedan referenciar los símbolos del kernel sys call table y sys ni syscall. Los pasos a seguir son:

• Copiar el archivo i386 ksyms 32.c (contenido en base2.zip) en el directorio del código fuente del kernel arch / x86 / kernel.
• Desde el directorio linux-4.7.5, compilar el kernel ejecutando el comando make.

1.1 Ejemplo 4: llamada al sistema vacía

Os facilitamos el código fuente de un módulo (modules / example4 / newsyscall.c) que añade un nuevo llamamiento 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 mesa tiene NR syscall entradas.

2. Muestra (utilizando la rutina printk) el índice (el número de posición) correspondiente a la entrada encuentro. Este índice será el identificador que se asociará a la nueva llamada al sistema. En condiciones normales, este índice será el 17.

3. Sustituir el contenido de la entrada no utilizada de sys call table por la dirección de memoria de una rutina (sys newsyscall, implementada al 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 en la tabla sys call table.

Para conseguir que un programa de usuario pueda invocar la nueva llamada al sistema a utilizar la llamada al sistema indirecta syscall (). Está parametrizada con el identificador numérico de la llamada al sistema tema que queremos invocar y la lista de parámetros de esta llamada al sistema. Por ejemplo, podríamos invocar la llamada read (fd, & c, 1) mediante syscall (NR read, fd, & c, 1). El programa de usuario modules / example4 / test4.c invoca la nueva llamada al sistema (asumiendo que se ha instalado en la posición 17).

El pedido make a la máquina host compila el módulo y el programa de pruebas. Los ficheros newsyscall.ko y test4 deben ser copiados en la máquina guest. Una vez instalado el módulo, la ejecución de test4 la máquina guest debería provocar 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 lınea de comandos.

Se muestra un ejemplo de la instalación y funcionamiento del módulo de ejemplo.

```
root@virtual:~/modules/example4#
root@virtual:~/modules/example4# insmod newsyscall.ko
New syscall installed. Identifier = 17
root@virtual:~/modules/example4# ./test4
Hello world, parameter 0
Return 29
root@virtual:~/modules/example4# ./test4 45
Hello world, parameter 45
Return 29
root@virtual:~/modules/example4# rmmod newsyscall
New syscall removed
root@virtual:~/modules/example4#
root@virtual:~/modules/example4#
```

1.2 Llamada al sistema a implementar

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

El siguiente paso será añadir a la llamada al sistema dos parámetros de entrada de tipo entero: el primero parámetro corresponde a un identificador de proceso y el segundo al tipo de operación. Si el valor del segundo parámetro es NUM CHILDREN (0), retornará el número de procesos hijos del proceso; si es NUM Siblingen (1) retornará el número de procesos hermanos del proceso. Si el proceso especificado no existe, retornará el código de error -ESRCH; si el tipo de operación no es correcto retornará el código de error -EINVAL.

Para calcular el número de procesos hijos y de procesos hermanos de un proceso es necesario acceder a su estructura de datos de tipo task struct que contiene el PCB del proceso. Esta estructura tiene un montón de campos, pero los relevantes en este ejercicio son children y sibling, los dos de tipo list head.
Cada uno de estos campos no es más que una lista doblemente encadenada que permite obtener la lista de procesos hijos y de procesos hermanos respectivamente.

También deberá utilizar la rutina del kernel struct task struct * find task by vpid (pid t VNR) que, dado un identificador de proceso devuelve el puntero a su struct task struct o NULL si el proceso no existe. Utilizando cscope (u otras herramientas similares, vistas en el práctica 1) es posible ver la definición y ejemplos de uso de list head y de find task by vpid.

El programa de usuario modules / newsyscall / testnew.c invoca varias veces la nueva llamada al sistema y realiza algunas pruebas para determinar su correcto funcionamiento. Si la ejecución del programa no llega hasta su última sentencia, indica que su módulo tiene 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 realizado (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]) Describe el concepto de driver (capıtulo 1) y de descriptor de dispositivo (capıtulo 5.3.2). Los drivers Linux utilizan una estructura de datos similar a un descriptor de dispositivo. Por lo tanto, la implementación de las llamadas al sistema de entrada / salida está estructurada en dos niveles: uno independiente del dispositivo y otro de dependiente del dispositivo.

El apartado 2.1 presenta el código fuente de un módulo que da de alta un nuevo driver. Una vez estudiado este código fuente, el apartado 2.2 plantea el ejercicio a resolver en esta práctica.

2.1 Ejemplo 5: driver

En el directorio modules / example5 se presenta el código fuente de un módulo (abc.c) que da de alta dinámicamente un nuevo driver Linux. Este driver devuelve ordenadamente las letras del alfabeto. Para probar el módulo se dispone de un programa (test5.c) que prueba las llamadas al sistema implementadas por el driver. el pedido 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 describiremos el código fuente del módulo con el driver. La rutina de instalación del módulo indica el sistema operativo que se quiere registrar un nuevo driver. Esto se realiza con la rutina register chrdev (). Esta rutina está parametrizada con el mayor del dispositivo, el nombre del dispositivo y con un puntero a una estructura del tipo struct file operations.

> • El mayor es un número entero que identifica el dispositivo. Los mayores utilizados están definidos en el archivo include / linux / major.h. Hay que elegir un identificador libre y asignar a la macro DRIVER MAYOR, archivo abc.c, este valor. Elegimos el 231.
> • El nombre del dispositivo no es más que un string que se utilizará cuando se listen los dispositivos activos (archivo / proc / devices).
> • El puntero a una estructura del tipo struct file operations permetr`a que el kernel conozca cuáles son las rutinas especıficas para acceder al dispositivo. La estructura tiene punteros a las operaciones open (), read (), write (), llseek (), ioctl (), ...especificas del dispositivo; todas estas rutinas tienen una interfaz

definida. Antes de invocar la rutina register chrdev (), es preciso inicializar la estructura.

Una vez instalado el módulo (y registrado el driver) en la máquina guest, hay que crear un archivo de tipo dispositivo que esté asociado al mayor que usa como primer parámetro a la rutina register chrdev (). Una posible forma de hacerlo es utilizando el comando mknod / dev / EX5 c 231 0 a la máquina guest. A partir de ese momento, todas las operaciones de entrada / salida (llamadas al sistema open (), read (), write (), ...) hechas sobre este archivo serán servidas por las rutinas especificadas al tercer parámetro de la rutina register chrdev ().

La rutina open () del dispositivo comprueba que el archivo haya sido abierto en modo O RDONLY (campo f flags del segundo parámetro de la rutina) y devuelve 0, en otro caso devuelve 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 recibe cuatro parámetros: un puntero a una estructura struct file, la dirección de memoria del buffer de usuario donde se escribirán 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 archivo. El segundo y el tercero 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 devuelve el número de caracteres que han sido leídos. Además, este driver de ejemplo limita a tres el número máximo de caracteres que se pueden leer en cada lectura.

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

Al desinstalar el módulo hay que desregistrar el driver utilizando la rutina unregister chrdev ().

Se muestra un ejemplo de la instalación y funcionamiento del módulo de ejemplo.

```
root@virtual:~/modules/example5#
root@virtual:~/modules/example5#
root@virtual:~/modules/example5# mknod /dev/ex5 c 231 0
root@virtual:~/modules/example5# insmod abc.ko
Correctly installed
 Compiled at Nov  3 2016 12:18:58
root@virtual:~/modules/example5# ./test5


Checking open...OK


Checking read...abcdefghijklmnopqrstuvwxyzOK


Checking lseek...OK
All correct
root@virtual:~/modules/example5# rmmod abc
Cleanup successful
root@virtual:~/modules/example5#
root@virtual:~/modules/example5#
```

2.2 driver a implementar

Una vez que haya entendido el código de ejemplo del driver presentado en la sección anterior, puede resolver el ejercicio siguiente.

Se pide que transforma el módulo del ejemplo 3 (lo que contabilizaba las llamadas al sistema) de forma que sea posible interaccionar con él mediante el archivo de dispositivo / dev / driver. Para hacerlo, puede aprovechar parte del código del driver de ejemplo abc.c.

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

- open (): Para poder acceder a los datos será preciso abrir el dispositivo utilizando la llamada al sistema open () en modo O RDONLY. Si se intenta abrir el dispositivo en cualquier otro modo que no sea O RDONLY, la llamada haur`a devolver el código de error EACC. En caso de que el dispositivo ya esté abierto, la llamada retornar`a el código de error EBUSY. Para garantizar que la implementación de esta llamada sea correcta, es preciso acceder en exclusión mutua a una variable utilizando, por ejemplo, las rutinas del núcleo DEFINE Semaphore, up, down; investigue como utilizarlas.
- read (): Para leer los datos caldr`a utilizar la llamada al sistema read (int fd, char * buf, int num).
  Interpretaremos el puntero de lectura / escritura asociado al archivo de dispositivo como la posición (en bytes) en la tabla callos utilizada en el ejemplo 3 para contabilizar las llamadas al sistema. El tercer parámetro de la llamada read () se interpretará como el número máximo de bytes a leer (como cada elemento de la tabla callos ocupa 4 bytes, este número deberá ser múltiplo de 4). El segundo parámetro de la llamada read () se interpretará como la dirección de memoria del espacio de memoria de usuario a partir de la cual se dejarán los contadores leídos de la tabla. La llamada retornará como resultado el número

de bytes que han sido leídos. Un efecto lateral de esta llamada será incrementar el puntero de lectura / escritura sobre la mesa.

• lseek ():
La llamada lseek modificará el puntero de lectura sobre la mesa de forma similar a como lo permite hacer sobre un archivo ordinario. De esta forma, podremos posicionarnos para leer los datos relativos a una llamada al sistema concreta. Admitiremos desplazamientos SEEK CUR y SEEK SET y SEEK END (el número de entradas de la tabla es NR syscall). Como el puntero de lectura indexa la tabla a nivel de bytes y la tabla contiene datos de tipo integer (4 bytes), las posiciones validas de este puntero serán múltiples de 4.

• ioctl (): La llamada ioctl admitirá dos pedidos (definidas a syscalls.h):
- SC OWNER: permitirá indicar el pid del proceso que se quiere monitorizar. Deberá especificar el pid del proceso como argumento del pedido.
- SC CLEAR: permitirá poner a cero el contador asociado a una llamada al sistema. Deberá especificar el identificador de la llamada al sistema (por ejemplo NR open) como argumento de la pedido.

• close (): La llamada close liberará el dispositivo de forma 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 devolver el código de error apropiada a cada caso.

• Puede utilizar la rutina printk para imprimir "chivatos" en el código del módulo.

• Se dispone de un programa de prueba (modules / driver / testdriver.c) para comprobar el correcto funcionamiento del driver. Para generar el ejecutable es necesario ejecutar make. Si el programa de prueba abortar antes de la última sentencia, significa que su driver no es correcto.

• Utilice la rutina printk para depurar el código del driver.

• En caso de duda respecto al comportamiento de las llamadas al sistema o los códigos de error que deben
devolver, consulte 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 realizado (formato pdf, incluyendo captura de pantalla del resultado del programa de pruebas).

Recursos

Recursos Básicos

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

recursos Complementarios

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