Taller de *drivers*

Sistemas Operativos

16 de mayo de 2024

1. Introducción

En este taller, construiremos tres sencillos *drivers* para el sistema operativo Linux. Es fundamental que tengan presentes los conceptos vistos en las clases teórica y práctica, que les deberían permitir responder a preguntas como:

- ¿Qué es un *driver*? ¿Qué función desempeña?
- ¿En qué se diferencia un *driver* de una pieza de software convencional?
- En un sistema UNIX, ¿cómo interactúan con un *driver* los usuarios?
- ¿Qué tipo de interfaz debe brindar el *driver*?

2. Módulos de *kernel*

Otra pregunta interesante para plantearse antes de continuar es si los *drivers* forman o no parte del *kernel*. En un sistema operativo de tipo UNIX (de *kernel* monolítico) podríamos decir que sí, ya los *drivers* ejecutan en modo de *kernel* y tienen acceso a todas sus estructuras y código. Sin embargo, es imposible que el *kernel* traiga soporte "de fábrica" para la enorme diversidad de dispositivos que podemos querer utilizar, y tampoco es deseable tener que recompilarlo cada vez que instalamos un nuevo dispositivo nuevo. Por eso, normalmente encontramos los *drivers* en forma de módulos que pueden acoplarse al *kernel* y extenderlo en tiempo de ejecución.

Un módulo debe contener el código que se ejecuta al cargar el módulo y al descargarlo del sistema, y también puede incluir código a ejecutarse para atender una llamadas al sistema (por ejemplo, el usuario solicita leer o escribir a un dispositivo) o para atender una interrupción (por ejemplo, una generada por un dispositivo para notificar que se completó una acción).

Los módulos de *kernel* corren con máximo privilegio, por lo que están completamente habilitados para causar desastres en nuestro sistema. Para evitar inconvenientes, haremos nuestros experimentos dentro de una instancia de la máquina virtual oficial de la materia. Por lo tanto, el primer paso es descargarla e instalarla siguiendo los pasos que se indican. No olviden configurar una carpeta compartida, de modo que puedan programar el código cómodamente en sus computadoras y solo tengan que usar la máquina virtual para compilar y hacer pruebas.

El archivo `ejemplo/hello.c` que acompaña a este taller sirve como ejemplo de un módulo muy sencillo. Allí pueden observar el uso de las macros `module_init` y `module_exit` (definidas en `<linux/module.h>`) para declarar qué funciones se ejecutarán al cargar y al descargar el módulo, respectivamente. Al final del archivo se utilizan otras tres macros para indicar la licencia, el autor y una breve descripción del módulo. Por último, también pueden observar el uso de la función `printk` (definida en `<linux/kernel.h>`), una suerte de análoga a `printf` que permite imprimir en el log del *kernel*; deben

tener en cuenta que dentro del *kernel* no se tiene acceso a la biblioteca estándar de C, y no existe tal cosa como "imprimir a la salida estándar".

El ejemplo viene acompañado de un `Makefile` que permite compilarlo; podrán observar que este proceso también es diferente de lo usual, ya que depende de la versión específica del *kernel* para la que quiera compilarse.

En Linux, se pueden ver los módulos cargados actualmente ejecutando `lsmod`. Para cargar un módulo se usa `insmod <nombre del módulo>`, y para descargarlo, `rmmod <nombre del módulo>`. Observen qué sucede para el módulo del ejemplo si, tras compilarlo (`make`), prueban cargarlo al *kernel* (`insmod hello.ko`) y luego descargarlo (`rmmod hello.ko`).

3. Preparando un *driver*

Para que un *driver* tenga alguna utilidad, tiene que poder ser accedido por los usuarios. En Linux esto se hace a través del sistema de archivos: los dispositivos (físicos o virtuales) aparecen representados como archivos dentro del directorio `/dev`. Si hacemos `ls /dev -l`, podemos ver algo como:

```
$ ls -l /dev
lrwxrwxrwx  1 root root           3 2010-10-08 20:00 cdrom -> sr0
...
crw-rw-rw-  1 root root      1,   8 2010-10-08 20:00 random
...
brw-rw----  1 root disk      8,   0 2010-10-08 20:00 sda
brw-rw----  1 root disk      8,   1 2010-10-08 20:00 sda1
...
```

El primer carácter de cada línea representa el tipo de archivo:

- `l` es un *symlink* (enlace simbólico).
- `c` es un *char device*.
- `b` es un *block device*.

Además, los *devices* tienen un par de números asociados:

- *major*: está asociado a un *driver* en particular (primer número luego del grupo).
- *minor*: identifica a un dispositivo específico que el *driver* maneja (segundo número luego del grupo).

Todo esto significa que un módulo se verá obligado a realizar algunos pasos "burocráticos" para figurar en el sistema. En particular:

1. Inicializarse como dispositivo. En el caso de este taller, nos ocuparemos exclusivamente de programar *char devices*, por lo que tendremos que hacer `#include <linux/cdev.h>` y luego llamar a la función

   ```
   void cdev_init(struct cdev *cdev, struct file_operations *fops);
   ```

   durante la inicialización del módulo.

   El primer parámetro es una estructura que representará al dispositivo. La segunda estructura, cuyo tipo se define en `<linux/fs.h>`, nos permitirá definir las operaciones que conformarán la interfaz del *driver*; volveremos aquí en breve. Deberemos definir ambas estructuras como `static` al comienzo de nuestro módulo.

2. Conseguir un *major* y un *minor*. Conviene pedirle al *kernel* que nos reserve el *major* de manera dinámica, para lo cual podemos usar

   ```
   int alloc_chrdev_region(dev_t *num, unsigned int firstminor,
       unsigned int count, char *name);
   ```

   El parámetro `num` corresponde al *major*, que nos será devuelto tras ejecutar la función. Los parámetros `firstminor` y `count` podemos ponerlos respectivamente en 0 y 1, con lo cual el *minor* será 0. `name`, por su parte, corresponde a un nombre para el dispositivo. Luego tendremos que asignar los números al dispositivo que inicializamos previamente, mediante

   ```
   int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
   ```

   Al descargar el módulo deberemos liberar estos números, llamando a

   ```
   void unregister_chrdev_region(dev_t num, unsigned int count);
   void cdev_del(struct cdev *dev);
   ```

3. Crear los nodos correspondientes en el sistema de archivos. Esto se hace desde el espacio de usuario, y podemos hacerlo manualmente por consola mediante el comando `mknod <nodo> c < major> <minor>`. Sin embargo, una opción más prolija es que sea el módulo quien se encargue de solicitar su creación.

   Para eso deberemos incluir `<linux/device.h>`, definir una variable estática `static struct class *mi_class` y luego, durante la inicialización del módulo, ejecutar

   ```
   mi_class = class_create(THIS_MODULE, DEVICE_NAME);
   device_create(mi_class, NULL, num, NULL, DEVICE_NAME);
   ```

   donde `DEVICE_NAME` es el nombre con el que figurará el dispositivo en el sistema de archivos y `num` es el *major*.

   A la hora de descargar el módulo, solicitaremos la destrucción de los nodos llamando a

   ```
   device_destroy(mi_class, num);
   class_destroy(mi_class);
   ```

4. Operaciones de archivos

Lo que le está faltando a nuestro módulo son las operaciones con que implementará la interfaz a la que accederán sus usuarios. Aquí es donde entra en juego la estructura `file_operations` que mencionamos antes:

```
struct file_operations {
    struct module *owner;
    ...
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*open) (struct inode *, struct file *);
    ssize_t (*release) (struct inode *, struct file *);
    ...
}
```

Cada uno de sus campos contiene un puntero a la función que se ejecutará cuando, desde el modo usuario, se realicen determinadas acciones sobre el archivo. Por ejemplo, las cuatro que aparecen detalladas corresponden a las operaciones `read`, `write`, `open` y `close`. Si algunos de los punteros se dejan en `NULL`, quedarán asociados a operaciones por defecto. Un ejemplo podría ser:

```
static struct file_operations mis_operaciones = {
    .owner = THIS_MODULE,
    .read = mi_operacion_lectura,
};

ssize_t mi_operacion_lectura(struct file *filp, char __user *data, size_t s,
   loff_t *off) {
    return 0;
}
```

Las cuatro operaciones detalladas reciben un parámetro de tipo `struct file *`, que es un puntero a una estructura representando el archivo abierto (es una distinta por cada vez que se haya invocado a `open`). Los otros parámetros de las funciones `read` y `write` son, en orden, el puntero al *buffer* en la memoria de usuario donde se quiere leer o escribir, la cantidad de bytes deseada, y el *offset* (que ignoraremos para el caso de *char devices*).

5. Usar los drivers

Podemos usar la terminal para poder hacer las operaciones de *read* o de *write*. Para hacer un *read*, usar el comando (seguramente debemos usarlo con *sudo*) *head* de la siguiente forma:

```
head -n cantBytesALeer /dev/nuestroDriver
```

Para el *write* podemos emplear una combinacion de *echo* y *tee* de la siguiente forma:

```
echo "textoAIngresarAlDriver" | sudo tee /dev/nuestroDriver
```

Si queremos hacer operaciones mas complejas como *open* o *close*, deberemos realizarlas con algun lenguaje de programacion, como *C*, *C++* o *Python*.

6. Nuestros primeros dispositivos

Ejercicio 1

Hechas las presentaciones, podemos pasar a resolver el primer ejercicio. La idea es basarse en el código de `ejemplo/hello.c`, extendiéndolo con las acciones recién descriptas para lograr implementar la funcionalidad pedida en el enunciado.

Se pide: Implementar un módulo `/dev/nulo` que replique exactamente la funcionalidad de `/dev/null`. Es decir: descartar toda la información que se escribe en él y, a su vez, no devolver ningún carácter cuando se intenta leer.

Ejercicio 2

Para el segundo ejercicio, deberemos tener en cuenta dos factores adicionales, relacionados con el manejo de memoria:

- Como ya discutimos en la clase práctica, al ejecutar en modo *kernel* no es seguro acceder directamente a la memoria del usuario. Deberemos hacerlo a través de las funciones

  ```
  unsigned long copy_to_user(void __user *to, const void *from, unsigned long
      count);
  ```

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long
    count);
```

que se encuentran definidas en `<linux/uaccess.h>`.

- Otra cosa que tampoco podemos hacer es pedir memoria dinámica de la forma usual, ya que la misma nos debe ser asignada en espacio de *kernel*. Para eso podemos usar las funciones

```
void * kmalloc(size\_t size, int flags); // Se suele usar la flag GFP_KERNEL.
kfree(void * ptr);
```

presentes en `<linux/slab.h>`.

Se pide: Escribir un módulo `/dev/azar` que debe comportarse de esta manera:

- Cada vez que el usuario invoca a `write`, se interpreta el contenido del *buffer* como un *string*, **al que hay que agregarle** un carácter de fin de *string* (`'\0'`), y el cual representará un número entero. Debe fallar con `-EPERM` si la entrada no se puede convertir a un número entero.
- Cada vez que el usuario lee del dispositivo, se devuelve una cadena de texto conteniendo un número al azar entre 0 y el valor que el usuario haya escrito previamente, y un carácter de fin de línea (`'\n'`). En caso de que no se haya escrito nada, se falla con `-EPERM`.

Funciones útiles: `kstrtoint` (`<linux/kernel.h>`), `get_random_bytes` (`<linux/random.h>`), `snprintf` (`<linux/kernel.h>`).

Ejercicio 3

Por último, en el tercer ejercicio deberemos comenzar a tener cuidado con posibles condiciones de carrera. Para eso nos será útil conocer los mecanismos de sincronización con los que contamos dentro del *kernel*, entre los que se encuentran:

- Semáforos (`struct semaphore`), con funciones como `sema_init(struct semaphore * sem, int val)`, `down(struct semaphore * sem)`, `down_interruptible(struct semaphore * sem )`, ..., `up(struct semaphore * sem)`.
- *Spinlocks* (`spinlock_t`), con funciones como `spin_lock_init(spinlock_t * lock)`, `spin_lock (spinlock_t * lock)`, `spin_unlock(spinlock_t * lock)`.

Además, vamos a tener que utilizar la estructura `file`. Recordar que la estructura se le pasa a cada función que opera sobre el archivo. Tiene varios campos, pero el más importante es el *private_data*. Es un puntero a *void*, con lo cual vamos a tener la habilidad de poder guardar cualquier cosa que deseemos y usarla entre las funciones. Es el equivalente a un atributo privado en programación orientada a objetos Ejemplo de uso:

```
typedef struct foo {
  char bar;
} foo;

static int your_driver_open(struct inode *inod, struct file *filp) {
  filp->private_data = kmalloc(sizeof(struct foo), GFP_KERNEL); // Reservamos
  memoria para la estructura
  ((foo *) filp->private_data)->bar = 4; // Como es puntero a void, necesitamos
   castearlo para poder usarlo
  return -EPERM;
}
```

```
static ssize_t your_driver_read(struct file *filp, char __user *data, size_t size
    , loff_t *offset) {
    foo *data = (foo *) filp->private_data; // Como es puntero a void,
    necesitamos castearlo para poder usarlo
    return data->bar
}
```

Se pide: Escribir un módulo `/dev/letras123`. El mismo posee tres espacios, inicialmente libres, que pueden ser ocupados por procesos. Su funcionalidad debe ser la siguiente:

- Cuando un usuario hace `open`, asignarle un espacio disponible si es posible, y de lo contrario fallar con `-EPERM`.
- Liberar el espacio asignado cuando el usuario haga `close` (o fallar con `-EPERM` si no se había realizado el `open` correspondiente).
- La primera vez que el usuario hace un `write`, si tiene un espacio libre asignado, guardar el primer carácter de la escritura. Los siguientes `write`s deben ser ignorados.
- Cada vez que el usuario lea del dispositivo, devolverle tantas copias del carácter guardado como haya pedido leer. Si no se hizo `write` previamente, fallar con `-EPERM`.

Por ultimo, cada ejercicio tiene una serie de tests asociados escritos en *Python*