Desarrollo de drivers en Linux

Manuel Ruiz López

1. Introducción

En este trabajo hablaremos sobre qué constituye a un driver, qué lugar ocupa dentro del sistema operativo y el funcionamiento de un driver en un sistema operativo moderno, concretamente en Linux. Seguidamente pasaremos a dar un par de ejemplos prácticos para ejemplificar el desarrollo de un de un driver de Linux.

2. Drivers: definición y uso

Un driver es el software encargado de controlar un periférico, ofreciendo una capa de abstracción sobre el hardware que permite al SO y las aplicaciones interactuar con el dispositivo mediante una interfaz estándar.

Su uso y funcionalidad no es universal y depende de las necesidades del sistema en concreto, En microcontroladores con recursos limitados y un hardware específico es usual que la propia aplicación desarrollada para el microcontrolador interactúe directamente con los periféricos a bajo nivel manipulando los registros de memoria y puertos. En proyectos de sistemas embebidos más complejos y ordenadores con sistemas operativos sencillos es usual que ésta tarea se desplace a los drivers, que abstraen la interacción a bajo nivel con el hardware ofreciendo una interfaz software a las aplicaciones para que éstas puedan hacer uso de los periféricos.

En este trabajo nos centraremos en el estudio de los drivers dentro de un sistema operativo moderno de uso general, en el que las aplicaciones de usuario no tienen permiso para interactuar directamente con los periféricos. En estos sistemas el kernel es el encargado de comunicarse con los diferentes drivers para soportar el uso de los distinos periféricos y a su vez ofrecer una interfaz estándar y común para que las aplicaciones puedan hacer uso de ellos.

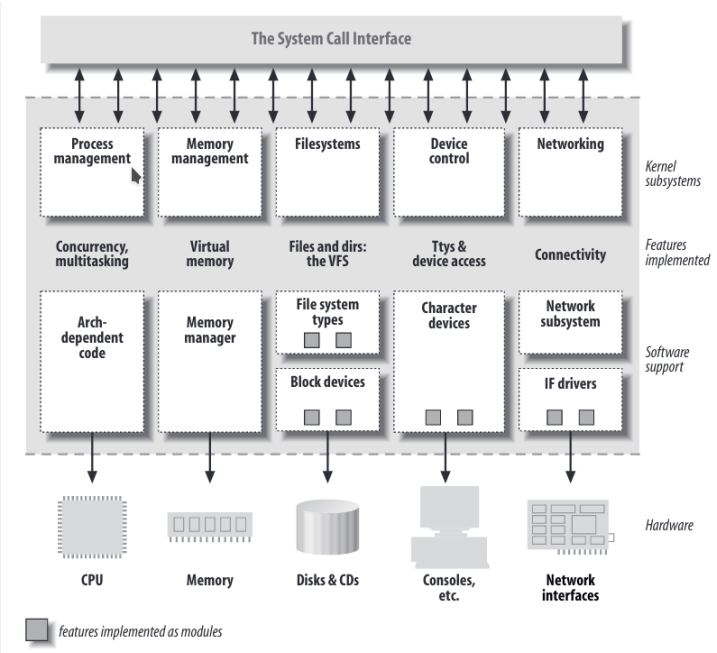
3. Arquitectura del kernel de Linux

Para entender cómo funcionan los drivers dentro de Linux primero hay que entender cual es la arquitectura de Linux y algunas de las características de su funcionamiento.

El kernel de Linux es un kernel monolítico, es decir, el SO al completo está contenido en un mismo programa que implementa la totalidad de funcionalidad que ofrece el kernel a las aplicaciones de usuario. En la imagen 1 podemos ver la arquitectura de Linux. Para evitar tener que generar un nuevo kernel y volver a iniciar el sistema cada vez que se quiera extender su funcionalidad existen los módulos.

En Linux tenemos la posibilidad de crear módulos software con nueva funcionalidad del kernel, compilarlos y cargarlos en el kernel en ejecución. No debemos confundir estos módulos con aplicaciones independientes o módulos de un SO no monolítico. Al cargar un módulo el kernel carga el código y los datos dentro de

Figura 1: Arquitectura del kernel de Linux [1]



su propio espacio de direcciones. De esta forma, podemos modificar la imagen del kernel en ejecución.

El kernel de Linux es el único programa dentro del SO que se ejecuta de forma privilegiada en la CPU (supervisor mode), ejecutándose el resto de aplicaciones en modo usuario. Al ser un kernel monolítico la totalidad de este se ejecuta en modo privilegiado y tiene acceso a todos los recursos del sistema. Para evitar vulnerabilidades de seguridad y evitar que un fallo en un programa en concreto derive en un mal funcionamiento del sistema en general el kernel acapara todos los recursos del sistema y el acceso a los dispositivos y se encarga de administrar y supervisar el acceso a estos recursos por parte del resto de aplicaciones.

Debido a estas restricciones los drivers del sistema deben estar incluidos dentro del kernel, para poder acceder directamente al hardware y controlar el acceso a estos por parte de los usuarios desde el SO. Esta arquitectura permite realizar una abstracción del hardware del sistema simplificando el uso de periféricos y recursos por parte de las aplicaciones de usuario.

4. Drivers en Linux

Una vez explicados algunos puntos claves de la arquitectura del kernel del Linux podemos pasar a ver las características de los drivers

Podemos encontrar tres tipos principales de drivers;

1. Character devices. Un dispositivo de caracteres es el que puede ser accedido como un flujo de bytes. Este comportamiento es definido por el driver, que implementa al menos las llamadas *open*, *close*, *read* y *write*. Este tipo de dispositivos son accedidos usualmente a través de nodos del sistema de ficheros. El driver de ejemplo que implementaremos en este trabajo será de este tipo.

2. Block devices. Un dispositivo de bloques es aquel que puede albergar un sistema de ficheros. Este tipo de dispositivos funcionan mediante operaciones de entrada/salida con un tamaño de bloque definido. En Linux el acceso a estos dispositivos por parte de las aplicaciones se realiza de la misma manera que los dispositivos de caracteres. La diferencia se encuentra en la forma que se manejan los datos de forma interna mediante el kernel/driver.

3. Network interfaces. Las interfaces de redes se encargan del envío y recepción de paquetes a otras anfitriones o interfaces y son controlados por el subsistema de networking del kernel. No son dispositivos con una interfaz orientada al flujo de datos por lo que ofrecen una interfaz diferente a los dispositivos de caracteres y bloques.

Como hemos comentado en el apartado anterior los drivers están contenidos dentro de la imagen del kernel. Esto implica que su ejecución se realiza en modo supervisor y que tienen acceso a los recursos del sistema y funcionalidad del kernel. Para poder ser incorporados a un kernel ya en funcionamiento se implementan en módulos. Al cargar este módulo se registra el driver en el kernel y se indica la funcionalidad que ofrece. De esta forma tenemos la posibilidad de que un driver se incluya en la compilación del kernel y sea inicializado al iniciar el sistema o que se cargue el módulo que lo implementa, siendo el resultado final el mismo: la funcionalidad del driver se encuentra en la imagen del kernel en funcionamiento.

Los drivers de Linux deben implementar una interfaz funcional estándar que depende del tipo de dispositivo que controlen. Esta interfaz define como se comporta el driver para cada una de las llamadas al sistema que implementa el kernel. De esta forma aunque el kernel albergue drivers con distinto hardware y comportamiento, esta complejidad intrínseca a cada dispositivo es manejada de forma interna y se ofrece al usuario una interfaz estándar común que permite el uso de dispositivos diferentes de forma transparente. Podemos encontrar información sobre cual es esta interfaz en la documentación del kernel sobre la API de los drivers [3].

Siguiendo la filosofía Unix en la que todo es un fichero, los dispositivos también son representados por el sistema como ficheros que se suelen encontrar en */dev/*. Estos ficheros almacenan que tipo de dispositivos son, que driver los maneja y un identificador del dispositivo en concreto, además de la información usual como los permisos. Estos ficheros sirven de representación software de un dispositivo hardware y ofrecen una forma sencilla de interacción para las aplicaciones de usuario que pueden interactuar con ellos de igual forma que con el resto de ficheros. Así se consigue que una aplicación que quiera mandar cierta

información a un dispositivo pueda hacerlo abriendo un fichero y escribiendo en el, realizándose de forma transparente al usuario la llamada al driver en cuestión, la identificación del dispositivo y las peculiaridades del hardware.

5. Desarrollo de un driver en Linux

Una vez explicada la arquitectura, funcionamiento y requisitos de los drivers pasamos a explicar el proceso de creación de un driver para Linux. Con este apartado pretendemos indicar los pasos necesarios para la creación de un driver básico que implemente una interfaz mínima, pero que ejemplifique las peculiaridades del desarrollo de un driver para Linux, sirviendo de base para la realización de un driver real. Para ampliar el conocimiento sobre este tema se recomiendan dos recursos bastante completos y que profundizan mucho más en el tema: *Linux Device Drivers* [1] y *The Linux kernel programming guide* [2].

La introducción al desarrollo de drivers en Linux la llevaremos a cabo mediante dos casos prácticos que servirán para ejemplificar la creación de módulos y el desarrollo de un driver tipo char respectivamente.

5.1. Caso práctico 1: Creación de un módulo

En *hello-1.c* podemos encontrar el código de este ejemplo, que implementa un módulo sencillo que recibe un parámetro y cuyo única funcionalidad es dejar un mensaje en el log del kernel al iniciarse y al terminar. Este código tiene tres secciones fundamentales. Al principio nos encontramos con cuatro líneas cuya finalidad es documentar el módulo:

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Manuel Ruiz Lopez");
MODULE_DESCRIPTION("Driver example");
MODULE_SUPPORTED_DEVICE("testdevice");
```

A continuación definimos una variable global, que servirá para almacenar el valor pasado por parámetro. Indicamos que parámetro vamos a recibir con la función *module_param*.

```
static char *name = "world";

module_param(name, charp, 0000);
MODULE_PARM_DESC(name, "Your name");
```

Finalmente llegamos a la parte fundamental del ejemplo donde implementamos la funcionalidad necesaria para la creación de un módulo del kernel:

```
static int __init hello_init(void)
{
```

```
    printk(KERN_INFO "Hello, %s\n", name);
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "Goodbye, %s\n", name);
}

module_init(hello_init);
module_exit(hello_exit);
```

Se definen dos funciones: una que implementa la lógica de inicialización del módulo, y otra que implementa la lógica a ejecutar cuando se elimine el módulo del kernel. Para indicar cuales son nuestras funciones de entrada y salida del kernel usamos *module_init* y *module_exit* respectivamente. La funcionalidad de nuestro módulo es mínima, dato que lo único que realiza es imprimir un mensaje de bienvenida y despedida en el log del kernel haciendo uso *printk*.

Una vez tenemos el código de nuestro módulo necesitamos compilarlo. Para realizar esta tarea debemos utilizar las herramientas que nos ofrece el kernel para el desarrollo de módulos. El uso de estas herramientas está definido en nuestro fichero *Makefile*:

```
KERNEL_VER=5.4.0-29-generic
obj-m += hello-1.o
obj-m += hello-2.o

all:
    make -C /lib/modules/$(KERNEL_VER)/build M=$(PWD)
        modules

clean:
    make -C /lib/modules/$(KERNEL_VER)/build M=$(PWD) clean
```

Al ejecutar *make* no solo se realiza la compilación del código si no que se genera el fichero *hello-1.ko*. Este fichero es un fichero objeto similar a un **.o* pero incluye información extra sobre el módulo. Esta información la podemos consultar ejecutando *modinfo hello-1.ko*. De esta forma podemos comprobar que se ha incluido la documentación que especificábamos al principio de nuestro código. Además podemos comprobar cómo está definida la versión del kernel para la que se ha compilado el código en el campo vermagic. Este campo define una seria de valores (números mágicos) que serán comprobados por el kernel a la hora de decidir si permite cargar el módulo o no.

Con el módulo ya compilado solo nos queda probar su ejecución. Podemos cargar el módulo y luego eliminarlo mediante los comandos *insmod hello-1.c* y *rmmod hello-1* respectivamente. Podemos modificar el valor del parámetro si cargamos el módulo mediante *insmod hello-1.c name=<my-param>*. Finalmente

podemos comprobar si todo ha funcionado correctamente comprobando si se ha registrado en el log del kernel los mensajes imprimidos por el módulo.

5.2. Caso práctico 2: Driver tipo char

En este ejemplo partimos del conocimiento adquirido en el ejemplo anterior para la creación de de un módulo y añadimos la funcionalidad necesaria para definir un driver sencillo de tipo char.

Implementamos cuatro operaciones en este driver: *open*, *release*, *read* y *write*. Indicamos con qué funciones implementamos estas operaciones mediante el *struct file_operations*.

Ahora nuestras funciones de entrada y salida del módulo deben modificarse para incluir el registro del dispositivo y la eliminación de este:

```
int init_module(void)
{
  Major = __register_chrdev(0, 0, 1, DEV_NAME, &fops);

  if (Major < 0) {
    printk(KERN_ALERT "Error in register_chrdev: %d\n",
        Major);
    return Major;
  }

  printk(KERN_INFO "I was assigned major number %d\n",
      Major);

  return 0;
}

void cleanup_module(void)
{
  __unregister_chrdev(Major, 0, 1, DEV_NAME);
}
```

Registramos un dispositivo de tipo char mediante la función *__register_chrdev* en el que solicitamos un número 'major' que identifique de forma unívoca al driver, solicitamos un rango de números 'minor' y le pasamos el nombre del dispositivo y las operaciones implementadas. Al pasarle como primer parámetro 0 le estamos indicando al kernel que nos asigne de forma dinámica el valor de 'major' que considere oportuno. Debemos comprobar que no se ha producido un erro al registrar el dispositivo. Al eliminar el módulo eliminamos el driver mediante la función *__unregister_chrdev*.

La funcionalidad del driver se encuentra implementada en las cuatro operaciones definidas. Es una funcionalidad sencilla que implementa las operaciones mínimas para que el dispositivo pueda ser escrito y leído. El driver tiene un buffer interno donde almacena un mensaje de saludo. Cuando se lee del dispositivo se

lee este mensaje de saludo y cuando se escribe en el se modifica a quién se dirige este saludo. A continuación se muestra la implementación de estas funciones:

```
static int device_open(struct inode * inode, struct file
    * file)
{
  if (Dev_Open) {
    return -EBUSY;
  }

  Dev_Open++;
  try_module_get(THIS_MODULE);

  return 0;
}

static int device_release(struct inode * inode, struct
    file * file)
{
  Dev_Open--;
  module_put(THIS_MODULE);

  return 0;
}

static ssize_t device_read(struct file * file, char *
    buffer, size_t length, loff_t* offset)
{
  int bytes_to_read;
  int bytes_read;

  if (BUF_LEN > length) {
    bytes_to_read = length;
  } else {
    bytes_to_read = BUF_LEN;
  }

  bytes_read = bytes_to_read - copy_to_user(buffer, msg,
      bytes_to_read);

  return bytes_read;
}

static ssize_t device_write(struct file * file, const
    char * buffer, size_t length, loff_t* offset)
{
```

```
  int max = BUF_LEN - 7;
  int bytes_to_write;

  if (max > length) {
    bytes_to_write = length;
  } else {
    bytes_to_write = max;
  }

  memset(&msg[7], 0, max);
  copy_from_user(&msg[7], buffer, bytes_to_write);

  return length;
}
```

Podemos observar como en *device_open* y *device_release* se implementa un mecanismo sencillo para comprobar si el driver ya está en uso haciendo de la variable *Dev_Open*. En las funciones *device_read* y *device_write* se realizan comprobaciones sobre la longitud a copiar y luego se realiza una copia de caracteres entre buffers. Es importan que esta copia de caracteres se realiza mediante las funciones aportadas por el kernel *copy_to_user* y *copy_from_user* pues el buffer del usuario se encuentre en el espacio de memoria de aplicación de usuario, al contrario que el buffer del driver que se encuentra en el kernel por lo que no debemos realizar la copia de datos de forma manual.

Para compilar y cargar el módulo realizamos los mismos pasos que en el ejemplo anterior. Podemos observar en el log del kernel si se ha realizado con éxito el registro del driver y el identificador que se le ha asignado.

Una vez cargado el módulo nuestro driver ya está disponible con la funcionalidad que hemos implementado, pero falta todavía un fichero que represente al dispositivo que queremos usar. Podemos crear este fichero de forma sencilla mediante la siguiente instrucción:

```
\$ mknod /dev/hello c <major> 0
```

De esta forma creamos un fichero de nombre *hello* que representa al dispositivo 0 gestionado por el driver tipo char con id *<major>*. Una vez creado ya podemos usar este fichero para interaccionar con nuestro dispositivo, realizando lecturas y escrituras. Si utilizamos, por ejemplo, el siguiente conjunto de instrucciones:

```
\$ head -c 50 /dev/hello
\$ echo "Paco" > /dev/hello
\$ head -c 50 /dev/hello
```

Obtendríamos como resultado primero la cadena 'Hello, world' y seguidamente 'Hello, Paco'. Estos resultados nos demuestran como podemos interaccionar con el fichero, de igual forma a cualquier otro fichero, mediante flujo de datos, y como realizando sencillas operaciones de lectura y escritura sobre el fichero, se están realizando las operaciones que definimos en nuestro driver.

Este ejemplo implementa un dispositivo que no atiende a un hardware real ni implementa una funcionalidad útil pero demuestra como se realiza el acoplamiento de un driver al kernel de Linux sirviendo de base para la realización de un driver de mayor complejidad.

Referencias

[1] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[2] Peter Jay Salzman, Michael Burian, Ori Pomerantz. The linux kernel module programming guide. `https://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html`, 2007.

[3] The kernel development community. The linux driver implementer's api guide. `https://www.kernel.org/doc/html/v4.14/driver-api/index.html`, 2020.