Instituto Tecnológico de Costa Rica
Área Académica de Ingeniería en Computadores
Introducción a los Sistemas Embebidos
Prof. Ing. Jeferson González Gómez

Tutorial: Introducción a drivers en Linux - I

Basado en: `http://www.freesoftwaremagazine.com/articles/drivers_linux`

Índice

1. Introducción

Este tutorial presenta un enfoque introductorio a la creación y uso de controladores de dispositivos (*drivers*) en Linux. En el desarrollo de este tutorial se creará un driver virtual para manejo de memoria, utilizando los métodos estándar de Linux para la inicialización, apertura, liberación, lectura, escritura y finalización en controladores de dispositivos.
A continuación se presentan algunos conceptos importantes a considerar:

Driver

Un *driver* es una aplicación de software que permite implementar rutinas para control y acceso a módulos y dispositivos de hardware. En Linux, la implementación del driver se realiza a partir de un archivo en lenguaje C, que, tras su compilación, generará un módulo de kernel (.ko), correspondiente al *driver* del dispositivo.

Espacios

Cuando se hacen uso de controladores de dispositivos, se debe hacer una diferencia entre los espacios o formas en que se puede realizar esto: espacio de kernel y espacio de usuario.

Espacio de kernel

El kernel maneja el hardware de una máquina de manera directa, simple y eficiente, brindando una interfaz de programación (API) uniforme. Cualquier subrutina o función que forma parte del kernel, como las encontradas en módulos y controladores de dispositivos, son considerados parte del espacio del kernel. La interacción con el hardware, en espacio de kernel, se realiza por medio de escritura y lecturas a direcciones de memoria, reservadas para el uso del hardware (mapas de memoria o puertos).

Espacio de usuario

Aplicaciones enfocadas al usuario, como los *shells* de UNIX, son parte del espacio de usuario. Estas aplicaciones no interaccionan directamente con el hardware, sino a través de funciones del kernel para tal fin, o llamadas al sistema (*syscalls*). La interacción con el hardware, en espacio de usuario, se realiza por medio de escritura y lecturas a archivos del sistema de archivos que provee el sistema operativo (para sistemas basados en UNIX).

Interfaz entre espacios

El acceso directo al hardware se realiza en espacio de kernel, sin embargo, el kernel debe proveer una interfaz para que las aplicaciones en espacio de usuario puedan también acceder a los diferentes dispositivos.

Usualmente, para cada función en espacio de usuario para el manejo de dispositivos como archivos, hay un equivalente en espacio de *kernel* que realiza escrituras y lecturas a direcciones de memoria (permitiendo la transferencia de información de espacio de *kernel* a usuario y viceversa). En el desarrollo de este tutorial se irán explicando cada una de las funciones disponibles en *drivers*, tanto en espacio de *kernel* como en espacio de usuario.

2. Cargar y remover driver en espacio de usuario

En este primer driver, se mostrará como cargar y remover un *driver* sin funcionalidad alguna. Para esto, se trabajará con un archivo *nothing.c* que tendrá el siguiente contenido:

```
#include <linux/module.h>

MODULE_LICENSE("Dual BSD/GPL");
```

Para la compilación del driver, se requerirá un archivo *Makefile*. El archivo Makefile deberá contener:

```
obj-m := nothing.o
```

Para versiones modernas del kernel de Linux, el *driver* debe compilarse contra una versión

específica del kernel de Linux, y sólo podrá ser cargado y usado en esa misma versión. Para compilar, por lo tanto, se requiere especificar la dirección donde se encuentra el código fuente del kernel (/usr/src/linux-headers-version-generic). El comando para compilar debe ser:

```
# make -C /usr/src/linux-headers-$(uname -r) M=$(pwd modules)
```

En caso tener error por no encontrar el código del kernel, este se puede instalar con:

```
# sudo apt-get install linux-headers-$(uname -r)
```

Entre otros archivos, la compilación deberá haber creado un archivo llamada *nothing.ko*, que corresponde al módulo del *driver*.

En **espacio de usuario**, se puede cargar el módulo con el comando *insmod*. Este comando permite la instalación del módulo dentro del kernel:

```
# sudo insmod nothing.ko
```

Para verificar que el módulo se haya cargado correctamente, se puede utilizar el comando *lsmod*:

```
# lsmod
```

Finalmente, el módulo puede ser removido del kernel, utilizando el comando rmmod:

```
# sudo rmmod nothing
```

Resumiendo, para cargar y remover un *driver* en **espacio de usuario**, se utilizan los comandos *insmod* y *rmmod*, respectivamente

3. Crear y remover un driver en espacio de kernel

Cuando se carga un *driver* en el kernel, se suelen realizar algunas tareas preliminares, como reservar espacios de memoria, puertos de entrada/salida, interrupciones, etc. Estas tareas se llevan a cabo en espacio de *kernel*, por lo que deben ser implementadas por el desarrollador explícitamente, para lo que se definen dos funciones en espacio de kernel: *module_init*() y *module_exit*(), que corresponden a los comandos en espacio de usuario *insmod* y *rmmod* respecticamente. Cuando el usuario ejecuta *insmod* o *rmmod*, el kernel debe llamar a *module_init*() y *module_exit*() según corresponda.

El siguiente ejemplo demuestra cómo definir una funcionalidad (simple) al cargar y remover un driver, en espacio de kernel. Se parte de un archivo *hello.c* que contiene:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
  printk("<1> Hello world!\n");
  return 0;
}

static void hello_exit(void) {
  printk("<1> Bye, cruel world\n");
}

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

En el código, las funciones *hello_init* y *hello_exit* pueden tener cualquier nombre, siempre y cuando, se pasen como parámetros a las funciones *module_init* y *module_exit*. La función *printk*, funciona de manera similar a *printf*, excepto que funciona a nivel de kernel.
Para compilar este módulo, se deberá modificar el Makefile de forma que se vea como:

```
obj-m := hello.o
```

El comando para compilar será el mismo de la sección anterior.
Al insertar (mediante *insmod*) el módulo del driver *hello.ko*, recién compilado, deberá llamarse a la función *hello_init* que imprimirá un mensaje en la **consola del sistema**, no en la terminal de *shell*. Para ver la consola del sistema puede utilizar el comando:

```
# dmesg | tail
```

$->$ Inserte y remueva el módulo *hello.ko* y verifique que se muestren los mensajes correspondientes.

En resumen, para insertar y remover un módulo de driver en **espacio de kernel**, se utilizan los métodos *module_init* y *module_exit*.

4. Driver completo: Memoria

En esta sección se desarrollará un *driver* completo para un "dispositivo"memoria, que permitirá escribir y leer un byte del mismo. Este ejemplo, no tiene mucha utilidad práctica, pero da un acercamiento ilustrativo a la creación de drivers, al ser un driver completo y fácil de implementar (no tiene interacción con hardware externo).
Para esta sección se parte de un archivo *memory.c* que se irá modificando para agregar funcionalidades. El archivo, inicialmente, deberá contar con el siguiente contenido:

```
/* Necessary includes for device drivers */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
```

```
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <asm/uaccess.h> /* copy_from/to_user */

MODULE_LICENSE("Dual BSD/GPL");

/* Declaration of memory.c functions */
int memory_open(struct inode *inode, struct file *filp);
int memory_release(struct inode *inode, struct file *filp);
ssize_t memory_read(struct file *filp, char *buf,
                    size_t count, loff_t *f_pos);
ssize_t memory_write(struct file *filp, const char *buf,
                    size_t count, loff_t *f_pos);
void memory_exit(void);
int memory_init(void);

/* Structure that declares the usual file */
/* access functions */
struct file_operations memory_fops = {
  read: memory_read,
  write: memory_write,
  open: memory_open,
  release: memory_release
};

/* Declaration of the init and exit functions */
module_init(memory_init);
module_exit(memory_exit);

/* Global variables of the driver */
/* Major number */
int memory_major = 60;
/* Buffer to store data */
char *memory_buffer;
```

En el código, se definen funciones para el driver. Estas funciones corresponden a insertar y remover el driver desde espacio de kernel (*memory_init* y *memory_exit*) ya mencionadas, así como para abrir (*memory_open*), cerrar (*memory_close*), escribir (*memory_write*) y leer (*memory_read*), que serán implementadas posteriormente en este tutorial.

En el código se define una estructura estándar *file_operations* que permite ligar las operaciones de espacio de usuario de abrir, cerrar, escribir y leer archivos con las funciones descritas arriba, en espacio de kernel (memory_open..., etc).

Finalmente se definen variables globales del kernel: *major number*, que corresponde a un número que utiliza el kernel para enlazar el dispositivo con su archivo correspondiente, y *memory_buffer*, que corresponde a un puntero a la región de memoria que se utilizará como almacenamiento para el driver, para escrituras y lecturas.

4.1. Ligar el dispositivo con su archivo

En Linux, los dispositivos se acceden desde espacio de usuario de la misma forma que se acceden los archivos ("todo en Linux es un archivo"). Estos archivos de dispositivos se encuentran dentro del directorio /dev/. Para ligar un archivo a un módulo de kernel se utilizan dos números: *major* y *minor*. De estos dos, el *major* es utilizado por el kernel para enlazar el archivo al driver, y el *minor* para uso interno del dispositivo. Para efectos de este tutorial, se trabajará solamente con el *major*. Para realizar en enlace entre en driver y el archivo del dispositivo, se debe crear primero este último archivo, siguiendo el comando:

```
# sudo mknod /dev/memory c 60 0
```

Donde el 60 representa el *major* y 0 el *minor*. El nombre del archivo (*memory*) representará al archivo dentro del driver, y el *major* enlaza el driver al archivo recién creado.

Dentro del *driver*, debe crearse en el enlace correspondiente con el archivo de dispositivo (/dev/-memory), para tal fin se utiliza la función *register_chrdev* . Esta función toma tres argumentos: *major*, el nombre del módulo, y la estructura estándar *file_operations*, declarada en el código inicial *memory.c*. Este enlace debe realizarse en la función de *memory_init* que es llamada al insertar el driver al kernel. La función *memory_init*, se definirá como sigue:

```
int memory_init(void) {
  int result;

  /* Registering device */
  result = register_chrdev(memory_major, "memory", &memory_fops);
  if (result < 0) {
    printk(
      "<1>memory: cannot obtain major number %d\n", memory_major);
    return result;
  }

  /* Allocating memory for the buffer */
  memory_buffer = kmalloc(1, GFP_KERNEL);
  if (!memory_buffer) {
    result = -ENOMEM;
    goto fail;
  }
  /*Set buffer to 0 by default */
  memset(memory_buffer, 0, 1);

  printk("<1>Inserting memory module\n");
  return 0;

  fail:
    memory_exit();
    return result;
}
```

En el código, se utiliza *kmalloc* para reserva de memoria del *buffer* dentro del espacio de kernel, de manera similar a lo que realiza la función *malloc*.

4.2. Remover el driver

Para remover el driver del kernel dentro de la función *memori_exit*, se debe llamar a la función *unregister_chrdev*, que liberará el *major* del kernel. La función *memory_exit* se debe ver:

```
void memory_exit(void) {
  /* Freeing the major number */
  unregister_chrdev(memory_major, "memory");

  /* Freeing buffer memory */
  if (memory_buffer) {
    kfree(memory_buffer);
  }

  printk("<1>Removing memory module\n");

}
```

En el código, además, se liberará el espacio de memoria reservado dentro del kernel.

4.3. Abrir el dispositivo como archivo

Abrir un archivo, en espacio de usuario se puede realizar de diferentes formas (fopen, etc). En espacio de kernel, esta operación se hace por medio del miembro *open* de la estructura *file_operations*. Este método será llamado cada vez que se haga una apertura, en espacio de usuario, del archivo de dispositivo /dev/memory. Para el caso del driver, el miembro *open* de la estructura, está ligado en la misma definición de la estructura, con la función *memory_open*. La definición de la función es un estándar y sus argumentos no serán tratados en este tutorial.
Cuando se abre un archivo, normalmente es necesario inicializar variables o resetear el dispositivo. Sin embargo, en este ejemplo simple no se realizará ninguna de estas acciones.
La función *memory_open* debe verse como:

```
int memory_open(struct inode *inode, struct file *filp) {

  /* Success */
  return 0;
}
```

4.4. Cerrar el dispositivo como archivo

En espacio de usuario, cerrar un archivo puede realizar con la función *fclose*. En espacio de kernel, esta operación se hace por medio del miembro *release* de la estructura *file_operations*. En el driver este miembro está ligado a la función *memory_release*. La definición de la función es un estándar y sus argumentos no serán tratados en este tutorial.
Cuando se cierra un archivo, normalmente es necesario liberar memoria utilizada y cualquier variable relacionada con la apertura del archivo. Sin embargo, en este ejemplo simple no se realizará ninguna de estas acciones.

La función *memory_release* debe verse como:

```
int memory_release(struct inode *inode, struct file *filp) {

  /* Success */
  return 0;
}
```

4.5. Leer del dispositivo

Para leer de un dispositivo, en espacio de usuario, suele usarse la función *fread* o similar. En espacio de kernel, se utiliza el miembro *read* de la estructura *file_operations*. En el caso del driver, este miembro está ligado con la función *memory_read*, que debe tomar como argumentos una estructura de tipo de archivo, un *buffer* (buf) que será retornado a la función de espacio de usuario (fopen, cat, etc) que haga uso del driver para lectura del dispositivo, un contador (count) con la cantidad de bytes a transferir y finalmente la posición de la cuál empezar la lectura del archivo (f_pos).
Para este caso simple, la función *memory_read* transfiere un único byte del buffer del driver (*memory_buffer*) hacia el buffer de espacio de usuario (buff), por medio de la función *copy_to user*. La función debe siempre retornar el número de bytes leídos.
La función *memory_read* debe verse como:

```
ssize_t memory_read(struct file *filp, char *buf,
                    size_t count, loff_t *f_pos) {
  /* Changing reading position as best suits */
  if (*f_pos == 0) {
    *f_pos+=1;
    /* Transfering data to user space */
    copy_to_user(buf,memory_buffer,1);
    return 1;
  } else {
    return 0;
  }
}
```

4.6. Escribir al dispositivo

Para escribir a un archivo dispositivo en espacio de usuario se utiliza una función como *fwrite* o similar. En espacio de kernel, se utiliza el miembro *write* de la estructura *file_operations*. En el driver, este miembro está ligado a la función *memory_write*, que debe tomar como argumentos una estructura de tipo de archivo, un *buffer* (buf) que contendrá el la información a escribir por medio de la función en espacio de usuario correspondiente, driver para lectura del dispositivo, un contador (count) con la cantidad de bytes a transferir y finalmente la posición de la cuál empezar la lectura del archivo (f_pos).
Para este caso simple, la función *memory_write* transfiere un único byte del buffer de espacio de usuario (buf) hacia el buffer de espacio de kernel (*memory_buffer*), por medio de la función *copy_from user*. La función debe siempre retornar el número de bytes escritos.

La función *memory_write* debe verse como:

```
ssize_t memory_write( struct file *filp, const char *buf,
                      size_t count, loff_t *f_pos) {
```

```
  copy_from_user(memory_buffer,buf,1);
  return 1;
}
```

4.7. Prueba del driver

El archivo fuente del driver debe incluir el código inicial, mostrado en la primera parte de esta sección, así como la implementación de las funciones memory_init, memory_exit, memory_open, memory_release, memory_read y memory_write.

El archivo Makefile deberá modificarse para el driver (cambiar hello.o por memory.o). El comando de compilación en el mismo de las secciones anteriores.

Antes de iniciar la prueba, deberán cambiarse los permisos del archivo /dev/memory. Para esto, se debe ejecutar:

```
# sudo chmod 666 /dev/memory
```

Para probar el driver, deberá insertarse el módulo, con *insmod*.

```
# sudo insmod memory.ko
```

Posteriormente, deberá hacerse una escritura de un byte al archivo de dispositivo /dev/memory. Una forma sencilla de hacer esto es con el comando *echo*, por ejemplo:

```
# echo -n H >/dev/memory
```

Luego de la escritura, podrá hacerse la lectura del dispositivo, por medio del comando *cat*, por ejemplo.

```
# cat/dev/memory
```

Siguiendo el ejemplo anterior, la salida del comando debería ser "H". Esto verifica el correcto funcionamiento del driver.

5. Evaluación

5.1. Teoría

- Explique los dos espacios existentes para el manejo y diseño de drivers. Para cada función de un driver especifique cómo se realiza en cada espacio.

5.2. Práctica

- Modifique el driver anterior para que soporte escrituras y lecturas al dispositivo virtual de un ancho de 32 bits.