PA 4 Reading

Linux Device Driver 3rd Edition, free online:

- Chapter 1 An Introduction to Device Drivers
- Chapter 2 Building and Running Modules
- Chapter 3 Char Drivers

The rest of this document contains excerpts and summarizes these three chapters.

Contents

1 - Introduction

> ... The Linux kernel remains a large and complex body of code, however, and would-be kernel hackers need an entry point where they can approach the code without being overwhelmed by complexity. Often, device drivers provide that gateway.
>
> Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized [system] calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and "plugged in" at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.
>
> There are a number of reasons to be interested in the writing of Linux device drivers. The rate at which new hardware becomes available (and obsolete!) alone guarantees that driver writers will be busy for the foreseeable future. Individuals may need to know about drivers in order to gain access to a particular device that is of interest to them. Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets. And the open source nature of the

> Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users.
>
> …
>
> You can also look at your driver from a different perspective: it is a software layer that lies between the applications and the actual device. …
>
> …
>
> Many device drivers, indeed, are released together with user programs to help with configuration and access to the target device. Those programs can range from simple utilities to complete graphical applications. …
>
> …
>
> The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendability.
>
> …
>
> *Character devices*
>
> A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (`/dev/console`) and the serial ports (`/dev/ttyS0` and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as `/dev/tty1` and `/dev/lp0`. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially.
> …
>
> — Chapter 1

Consider reading the whole chapter to gain a better understanding of Linux device drivers.

2 - Char Device Driver

You are given a simple character device driver, based on the device driver developed in LDD Chapter 3, and a user-space program that uses the driver.

LDD Ch.3 explains in detail the construction of the driver. Here we summarize the key points.
Consider reading the whole chapter to gain a better understanding of Char Device Drivers.

The `scull` device driver is a Linux kernel module (lkm). The purpose of this driver is simply to store an integer, which can be manipulated from user-space through a set of well-defined operations.

2.1 - Major and Minor Numbers

> Char devices are accessed from user-space through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the `/dev` directory. Special files for char drivers are identified by a "c" in the first column of the output of `ls --l /dev`. Block devices appear in `/dev` as well, but they are identified by a "b". ...
>
> If you issue the `ls --l /dev` command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. ...
>
> The following listing shows a few devices as they appear on a Mac OS system (device drivers and special fields are not specific to Linux). Their major numbers are 25, 0, 14, 21, and 21, while the minors are 0, 0, 0, 7, and 13.

```
crw-rw-rw- 1 root wheel 25, 0 Mar 8 17:28 dtracehelper
crw-rw-rw- 1 root wheel 0, 0 Mar 8 17:28 fbt
crw-r--r-- 1 root wheel 14, 0 Mar 8 17:28 fsevents
crw-rw-rw- 1 root wheel 21, 7 Mar 12 12:58 io8log
crw-rw-rw- 1 root wheel 21, 13 Mar 12 13:55 io8logtemp
```

> Traditionally, the major number identifies the driver associated with the device. For example, `/dev/io8log` and `/dev/io8logtemp` are both managed by driver 21, whereas `/dev/fsevents` is managed by driver 14. Modern Linux kernels allow multiple drivers to share major numbers, but most devices that you will see are still organized on the one-major-one-driver principle.
>
> The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how your driver is written, you can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices. Either way, the kernel itself knows almost nothing about minor numbers beyond the fact that they refer to devices implemented by your driver.
>
> — Chapter 3

2.2 - Internal Representation of Device Numbers

Within the kernel, the `dev_t` type (defined in `<linux/types.h>`) is used to hold device numbers—both the major and minor parts. Your code should make use of a set of macros found in `<linux/kdev_t.h>` to manipulate them. To obtain the major or minor parts of a `dev_t`, use:

```c
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

If, instead, you have the major and minor numbers and need to turn them into a `dev_t`, use:

```c
MKDEV(int major, int minor);
```

— Chapter 3

2.3 - Allocating and Freeing Device Numbers

One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. The necessary function for this task is `register_chrdev_region`, which is declared in `<linux/fs.h>`:

```c
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

...

`register_chrdev_region` works well if you know ahead of time exactly which device numbers you want. Often, however, you will not know which major numbers your device will use; there is a constant effort within the Linux kernel development community to move over to the use of dynamically-allocated device numbers. The kernel will happily allocate a major number for you on the fly, but you must request this allocation by using a different function:

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

...

Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

```c
void unregister_chrdev_region(dev_t first, unsigned int count);
```

> The usual place to call `unregister_chrdev_region` would be in your module's cleanup function.
>
> — Chapter 3

The device driver you are given can use both methods. This is determined by the initial value of scull_major (a value of 0 triggers the dynamic allocation of the major number).

2.4 - Setting up Special Files

User-space can interact with devices through special files, by performing operations like open, read, write, close, etc. These operations are routed by the kernel to the driver that has registered the file's major and minor numbers. *Special files need to be created separately by the `root` user, they are not created automatically by the module when loading the module into the kernel.* For a given device driver with fixed known major / minor numbers, you can create the corresponding special file using the `mknod` command:

```
mknod filename device_type major minor
```

For example, `sudo mknod /dev/scull c 492 0` will create the special character device file `/dev/scull` with major / minor numbers 492 / 0 (this is an example only; see the next paragraph below to find out the correct number that you should use to replace 492). Note that the file will disappear when you reboot the kernel.

For dynamically assigned major numbers, you can find in the file `/proc/devices` the current major number assigned by the kernel after the module has been loaded. This major number might change every time you reboot the kernel! A typical `/proc/devices` file looks like the following:

Character devices:

```
1 mem
2 pty
3 ttyp
4 ttyS
6 lp
7 vcs
10 misc
13 input
14 sound
21 sg
180 usb
```

Block devices:

```
2 fd
8 sd
11 sr
65 sd
66 sd
```

2.4.1 - Registering the Device-Driver File Operations

The file operations performed on the device special file will be routed to the device driver. Before this happens, the driver needs to register itself with the kernel. This registration should include a list of the file operations supported by the driver, in the form of a list of function pointers pointing to the driver functions implementing them (*`file_operations` structure*). Of course, generic file operation handling is also performed by the kernel, the device driver function is the last one in a chain of functions triggered, when a user-space process performs an operation on the device's special file (for example, opens it).

To register the character device, your code should include `<linux/cdev.h>`, where the structure and its associated helper functions are defined.

There are two ways of allocating and initializing one of these structures. If you wish to obtain a standalone cdev structure at runtime, you may do so using `cdev_alloc`:

```c
struct cdev *my_cdev = cdev_alloc();


my_cdev->ops = &my_fops;
```

Chances are, however, that you will want to embed the `cdev` structure within a device-specific structure of your own; that is what the given driver does. In that case, you should initialize the structure that you have already allocated with `cdev_init`:

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

Either way, there is one other `struct cdev` field that you need to initialize. Like the `file_operations` structure, struct cdev has an owner field that should be set to `THIS_MODULE`.

Once the cdev structure is set up, the final step is to tell the kernel about it with a call to `cdev_add`:

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

There are a couple of important things to keep in mind when using `cdev_add`. The first is that this call can fail. If it returns a negative error code, your device has not been added to the system. It almost always succeeds, however, and that brings up the other point: as soon as `cdev_add` returns, your device is "live" and its operations can be called by the kernel. You

should not call `cdev_add` until your driver is completely ready to handle operations on the device.

To remove a char device from the system, call `cdev_del`:

```c
void cdev_del(struct cdev *dev);
```

Clearly, you should not access the cdev structure after passing it to `cdev_del`.

2.5 - File Operations

The `file_operations` structure (pointed to by `my_cdev->ops`) is how a char driver sets up this connection. The structure, defined in `<linux/fs.h>`, is a collection of function pointers. Each open file is associated with its own set of functions (by including a field called `f_op` that points to a file_operations structure). The operations are mostly in charge of implementing the system calls and are therefore, named `open`, `read`, and so on. We can consider the file to be an "object" and the functions operating on it to be its "methods," using object-oriented programming terminology to denote actions declared by an object to act on itself.

Conventionally, a `file_operations` structure or a pointer to one is called `fops` (or some variation thereof). Each field in the structure must point to the function in the driver that implements a specific operation, or be left `NULL` for unsupported operations. The exact behavior of the kernel when a `NULL` pointer is specified is different for each function, as the list later in this section shows.

The device driver given implements only a subset of device methods. Its `file_operations` structure is initialized as follows:

```c
struct file_operations scull_fops = {
  .owner = THIS_MODULE,
  .unlocked_ioctl = scull_ioctl,
  .open = scull_open,
  .release = scull_release,
};
```

Other important members include `read`, `write`, and `llseek`.

In the given module, `open` and `release` (corresponds to `close`) do nothing but print a message in the kernel log. I/O control, or `ioctl` for short, implements a variety of device commands, which will be discussed below.

2.5.1 - IOCTL

I/O control (IOCTL) operations are used to communicate with the driver, when common file operations (like `read` / `write`) are not sufficient. Applications can use the `ioctl` function to issue requests to a special file, corresponding to a device driver.

```c
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);
```

`fd` corresponds to the file descriptor of a special file, and request corresponds to a request (or command) code. There is an optional third argument which is commonly used to pass a pointer or long to the device driver.

IOCTL requests are routed to the device driver module, calling the handler setup using `scull_fops`:

```c
static long scull_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);
```

- `filp` corresponds to a pointer to struct file in the kernel (aka "file pointer").

We'll consistently call the pointer `filp` to prevent ambiguities with the structure itself. Thus, file refers to the structure and `filp` to a pointer to the structure.

- `cmd` corresponds to the request code passed by user-space.
- `arg` corresponds to the optional third argument passed by user-space.

The `ioctl` handler uses the request number to demultiplex requests. The optional `arg` can be used to pass data to the kernel, either directly (by passing a long) or indirectly (by passing a pointer to user-space data). A pointer can be also used to transfer data from kernel to user-space.

Request numbers follow a convention, so they should be defined using one of the following macros: `_IO`, `_IOW`, `_IOR`, or `_IOWR`. `scull.h` already includes the definition of a set of `ioctl` request numbers.

```c
#define SCULL_IOC_MAGIC 'k'

#define SCULL_IOCRESET _IO(SCULL_IOC_MAGIC, 0)

#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC, 1, int)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC, 2)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC, 3, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC, 4)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 5, int)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC, 6)
```

The first two arguments (code & sequence) are always bytes (char). While not necessary, the convention is that the two bytes form a unique combination. Requests that include a data

transfer, meaning that a pointer is passed (instead of a long), and a data copy operation needs to be performed, include a third argument that is the type of the data that is to be copied.

Consider the following two requests that both pass an integer value to the kernel:

- `SCULL_IOCTQUANTUM`: the third argument is used to directly pass an int (cast to a long) to the device driver.
- `SCULL_IOCGQUANTUM`: the argument is used to pass a pointer to an int (stored in user-space).

Depending on the direction of the data being transferred a different MACRO needs to be used.

- `_IOW` corresponds to a write operation from user-space, meaning that the kernel reads from user-space.
- `_IOR` corresponds to a read operation from user-space, meaning that the kernel writes to user-space.
- `_IOWR` corresponds to a write/read operation.
- `_IO` is used when no data needs to be copied.

When these macros have been used to properly configure the IOCTL command / request number, the handler can use a set of other macros to easily establish direction, data transfer size, etc.

Example:

```c
/*
 * extract the type and number bitfields, and don\'t decode
 * wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
 */

if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

/*
 * the direction is a bitmask, and VERIFY_WRITE catches R/W
 * transfers.
 `Type' is user-oriented, while
 * access_ok is kernel-oriented, so the concept of "read" and
 * "write" is reversed
 */

if (_IOC_DIR(cmd) & _IOC_READ)
  err = !access_ok_wrapper(VERIFY_WRITE, (void _user \*)arg, _IOC_SIZE(cmd));

else if (_IOC_DIR(cmd) & _IOC_WRITE)
  err = !access_ok_wrapper(VERIFY_READ, (void _user \*)arg, _IOC_SIZE(cmd));

if (err) return -EFAULT;
```

`access_ok_wrapper` is our own wrapper for `access_ok`. It simply checks that a user pointer, actually points to user-space, and some other lightweight checks. Because this check is performed first, the handler can then use `__get_user()` and `__put_user()` to copy (instead of using the `get_user()` and `put_user()` functions described in the "System Calls" lecture notes) as some of the checks have already been done by `access_ok`.