Operating Systems 2

Character and Block Devices

Arkadiusz Chrobot

Department of Computer Science

May 18, 2020

Outline

Introduction

One of the tasks that the VFS does is I/O devices handling. The word "device" doesn't have to mean a real hardware, it can be also a virtual device or a pseudo-device. Unix-like operating systems recognize three categories of devices — character, block and network devices — which are accessible to the user-space software. Some of them, like Linux, also use some subcategories, yet they are internal to the kernel. This lecture is about how the Linux kernel handles devices belonging the first two categories. More advanced topics concerning this subject, like the kernel device tree or the kernel device model are not discussed here.

Introduction

Hardware Infrastructure

All computer systems have some hardware infrastructure that allows them to communicate with external devices. One of the most popular, although far from perfect is the is the connection structure of the 32-bit computers based on x86 CPU family. Each device is attached to the computer with a help of an I/O bus, that consists of three parts: the data bus, the control bus and the address bus. The CPUs from the Pentium family use 16 or 32 address bus lines or 8, 16, 32 or 64 lines of the data bus. The bus is not directly connected to the device, but there is a hardware structure that mediates between them. This structure has up to three components: I/O *ports*, the *interface* and/or *controllers*. The ports are a set of addresses that are assigned to the devices. The x86 CPUs can address up to 65536 8-bit ports, but the ports can also be merged together into 16 or 32-bit units. The CPUs have special instructions to handle the ports, but they also allow to map those ports into the memory.

Introduction

Hardware Infrastructure

The Mapping is preferred, because it allows for DMA transfers, is more efficient and doesn't require special instructions to handle the ports. Some CPUs, for example those manufactured by the Motorola Company, use only this solution. The I/O ports are organized into sets of registers that enable the communication with devices. Typically there are four types of registers: the *status register*, the *control register*, the *input* register and the *output* register. In some cases a single register has two purposes. For example in the keyboard the status register is also the control register and the input register is the output register. The IØ interface is a hardware that converts the value in control register into the device instruction and detects the device state changes and modify accordingly the status register. Additionally it is attached to the PIC and triggers the interrupt on behalf of the device. There are two types of interfaces: general-purpose and specialized.

Introduction

Hardware Infrastructure

The former can service several kinds of devices attached to the system with the use of such buses like parallel, serial USB, PCMCIA, SCSI, etc. The latter are associated with a specific type of device, like the keyboard or mouse. More advanced devices require a controller that translates the high-level instruction send by the I/O interface into a series of microinstructions that can be interpreted by the device. It also changes the state of registers according to the signals that it receives from the I/O device.

Introduction

I/O Device Handling in Linux — General Description

In Unix-like operating system the character and block devices are handled the same way as files, i.e. with the use of the same system calls. They are also represented by files, which aside from the name have three additional attributes: *type* that specifies if the represented device is a character device or a block device, the *major* number and the *minor* number. Inside the kernel those numbers are combined into one 32-bit *device number* of the `dev_t` type. Starting from the series 2.6 of the kernel the major number occupies the most significant 20 bits of the device number, and the minor number occupies the least significant 12 bits. However, those numbers inside the device number should be always accessed with the use of the `MAJOR` and `MINOR` macros. They also ought to be merged into the device number with the use of the `MKDEV` macro. The reason for that is that in the earlier versions of the kernel the size of the major and the minor number was 16-bits. It has changed in the 2.6 series and so it may change in the future kernel releases.

Introduction

I/O Device Handling in Linux — General Description

The major number identifies the driver that is responsible in the kernel for handling a family of devices (like the printers for example). The minor number identifies the specific device handled by the driver. It is useful when more than one device of a given family is attached to the computer. The drivers can be implemented as an immanent part of the kernel or in a form of kernel module.

Character Device Drivers

The character devices usually provide a sequential access to the data and transfer a relatively small portions of information, like several bytes. Moreover the size can be different in each transfer. An example of the character device can be a keyboard or a mouse. The first thing that the character device driver does is acquiring one or several of major numbers with the help of the following function:

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

The `first` parameter specifies the first device number from a pool of such numbers that should be acquired. Should the driver be available to all Linux users then the numbers it uses have to be assigned by *The Linux Assigned Name and Numbers Authority* (www.lanana.org). Otherwise the availability of those numbers can be verified in the `/proc/devices` file or in the `/sys` directory. The `count` parameter specifies the number of device numbers that have to be allocated and the `name` parameter the string of characters that represents the name of the device.

Character Device Drivers

The function returns 0 if it manages to successfully acquire the device numbers. More convenient to use is the following function:

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

It allocated to the driver a specified number of device numbers starting with the first available device number. The programmer doesn't specify the first device number. The `dev` parameter is an output parameter. The function uses it to return the first allocated device number. The `firstminor` specifies the value of the first minor number that should be allocated. Usually it is 0. The last two parameters are the same as in the `register_chardev_region()` function. If successful the function returns 0. If the device numbers are no longer used they should be unregistered with the use of the following function:

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

Character Device Drivers

The character device drivers use three of the VFS structures: the file object, the file method table and the i-node object. The file method table should contain addresses of the functions that perform operations on the device file. If the driver is implemented as a method table then the value of the `THIS_MODULE` macro should be assigned to its `owner` field. It prevents unloading the module when one of the methods is performed. Usually the programmers who write the device drivers implement four methods: `open()`, `read()`, `write()` and `close()`, although implementing all of them in one driver is not necessary. If the device requires some specific operations that cannot be provided by those four functions then one of the `ioctl()` methods has to be implemented. The other methods can be left unimplemented. The driver may make use of the following members of the file object: `mode` — stores the access permissions, `f_pos` — it is the file pointer, `f_flags` — stores flags, `f_ops` — points to the method table, `private_data` and `f_dentry` — points to the dentry object.

Character Device Drivers

The `mode` field may be verified by the `open()` method, but it is not necessary, because it is checked by other parts of the kernel, before this method is called. The driver checks the flags field to decide if the operations have to be synchronous or asynchronous. The 64-bit value of the `f_pos` field is passed to the `llseek()` method, which returns modified value of the file pointer. Also the `read()` and `write()` methods use this pointer, which is passed to them by their last parameter. The `private_data` field is a pointer of the `void *` type, that can point to a dynamically allocated memory area used for storing data that shouldn't be lost between methods calls. The memory area should be allocated by the first invocation of the `open()` method, and deallocated by the invocation of the `release()` method that follows the last invocation of the user-space `close()` function. The `f_dentry` field is used for acquiring the address of the i-node object.

Character Device Drivers

In the i-node object the driver can use the `i_rdev` field that stores the device number. To obtain the major and minor number from that field the following macros can be used:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

Other field of this object is the `i_cdev` pointer which points to a structure that represents the character device served by the driver in the kernel. The structure has to be created and initialized with the use of the `cdev_alloc()` function or it can be statically allocated and initialize with the help of the following function:

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

In both cases the address of the structure has to be stored in the `i_cdev` field and the value of the `THIS_MODULE` macro has to be assigned to its `owner` field. Also when the structure is created with the use of the `cdev_alloc()` function, the `ops` field of the `cdev` structure has to be initialized directly.

Character Device Drivers

After the `cdev` structure is created it has to be added to other such structures stored by the kernel with the help of the following function:

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

This function removes the `cdev` structure from other such structures stored by the kernel:

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

Each device handled by the driver has to have its own `cdev` structure. In the earlier releases of the kernel the driver didn't have to create such a structure. The device was registered with the help of the `register_chrdev()` function and unregistered by the `unregister_chrdev()` function. Starting for some releases of the 2.6 kernel series the device model subsystem has to be informed about a new driver. It happens when the driver is initialized and requires using a macro and a function. The macro creates a structure that describes the class of the device handled by the driver.

Character Device Drivers

The declaration of the macro is as follows:

```
class_create(owner, name);
```

Its first argument is the value of the `THIS_MODULE` macro, and the second is the name of the class. The function is declared as follows:

```
struct device *device_create(struct class *class, struct
  device *parent, dev_t devt, void *drvdata, const char
                    fmt, …);
```

It creates and registers int the `sysfs` file system a structure that represents the device. Its fist argument is the address of the class structure. The second argument is the address of a parent data structure — it can be `NULL` if no such structure exists. The third argument is the device number. The fifth argument is a pointer to a data stored in the structure and used by callback functions — it also can be `NULL`. The fifth argument is a string that represents the name of the device. It can contain formatting strings, just like the `printf()` function.

Character Device Drivers

The structure created by the `device_create()` function can be removed with the use of the following function:

```
void device_destroy(struct class *cls, dev_t devt);
```

It has two arguments: the address of the class structure and the device number. The class structure can be then freed using the following function:

```
void class_destroy(struct class *cls);
```

As an argument it takes the address of the released class structure. The behaviour of the device driver methods must follow a specific protocol. The `open()` method should:

- identify the device the driver handles — get the minor number,
- check if there are no errors specific for that device,
- initialize the device, if it is opened for the first time,
- update file pointer, if necessary,
- allocate and initialize the memory area for private data, if necessary.

Character Device Drivers

Likewise the `release()` method should follow this protocol:

- deallocate the memory area for the private data, if it has been allocated by the `open()` method,
- shut down the device after the last invocation of the user-space `close()` function.

The implementations of `read()` and `write()` methods should also respect some rules. They should return the number of actually read/written bytes. In case of failure they should return an error code that identifies the cause, like `-EINTR` — a signal has been received, `-EFAULT` — a bad address, `-EIO` general input-output error. For more detailed description of the character device driver API please refer the eight laboratory instruction.

Block Device Drivers

The block device drivers use similar structures and operations as character device drivers. However, the handling of block devices is a more challenging task, and some of its details will be discussed in the next lecture. The block devices provide random access to data and they transfer them in portions called *blocks*, hence the name of those devices. The size of a single block is a even multiple of the *sector* size. The kernel assumes that the size of the sector is 512 bytes.

The first thing that the block device driver does when initialized is acquiring a major number with the help of the `register_blkdev()` function, which is declared in the `linux/fs.h` header file in the following way:

```
int register_blkdev(unsigned int major, const char *name);
```

If the first argument of this function is 0 then it will allocate the first available major number.

Block Device Drivers

The allocated major number can be released using the following function:

```
void unregister_blkdev(unsigned int major, const char *name);
```

The block device drivers have their own method table which is a structure of the `struct block_device_operations` type declared in the `linux/blkdev.h` header file. It has the `owner` field and several other members that should point to such methods as: `open()`, `release()`, `ioctl()`, `compat_ioctl()`, `check_events()` and finally `revalidate_disk()`. The `check_events()` method is invoked mainly when the medium in the device is changed and it invocation is followed by the call to the `revalidate_disk()` method.

Just like a character device is represented by the `cdev` structure the block device is represented by a structure of the `struct gendisk` type, which is declared in the `linux/genhd.h` header file.

Block Device Drivers

This structure has the following members: `major` — stores the major number, `first_minor` — stores the first minor number, `minors` — stores the number of minor numbers, `disk_name` — stores a string that represents the name of the device (up to 32 characters), `fops` — stores the address of the `block_device_operations` structure, `queue` — stores the address of the *request queue*, `flags` — stores flags (rarely used, usually only for pluggable devices and optical disks) and `private_data` — points to the memory area that stores driver's private data. The `gendisk` structure also stores the capacity of the block device, expressed in sectors. This value is set with the help of the `set_capacity()` function. The `gendisk` structure is allocated with the use of the `alloc_disk()` function and released after its reference counter reaches zero with the help of the `put_disk()` function:

```
struct gendisk *alloc_disk(int minors);
void put_disk(struct gendisk *disk);
```

Block Device Drivers

Each `gendisk` structure represents a single device handled by the driver, for example a single partition of the hard disk. To make the device available to the rest of the kernel, the driver should call the `add_disk()` function for its `gendisk` structure:

```
void add_disk(struct gendisk *gd);
```

The structure can be removed with the use of the `del_gendisk()` function:

```
void del_gendisk(struct gendisk *gd);
```

The most important member of the `gendisk` structure is the `queue` field, which point to the request queue. The memory for the queue is allocated with the use of the `blk_init_queue()`:

```
request_queue_t *blk_init_queue(request_fn_proc
        *request, spinlock_t *lock);
```

The first argument of this function should be an address of a function that processes the requests from the queue and the second ought to be the address of a spin lock that protects the queue.

Block Device Drivers

If the driver services a device that unlike hard disks offers a truly random access to data (for example a flash memory device), then the request queue is redundant. In this case the `queue` field of the `gendisk` structure is initialized with the use of the `blk_alloc_queue()` function:

```
request_queue_t *blk_alloc_queue(int flags);
```

If such an initialization is performed then the driver should provide an implementation of the `make_request()` function that processes a single request. The function should be registered with the use of the following function:

```
void blk_queue_make_request(request_queue_t *queue,
                            make_request_fn *func);
```

The request queue is deallocated with the use of the following function:

```
void blk_cleanup_queue(struct request_queue *q);
```

For more detailed description of the block device driver API please refer the ninth laboratory instruction.

Questions

?

The End

Thank You for Your attention!