Laboratory 8: "Character device drivers" (two weeks)

Arkadiusz Chrobot, PhD
Karol Tomaszewski, PhD

May 18, 2024

Contents

Introduction **1**

1. Character device drivers **1**

2. Description of character device drivers API **2**

3. The `Udev` configuration **9**

4. Example **9**

Exercises **12**

Introduction

On Linux, just like on other Unix-compatible systems, there are three main device categories distinguished:

character devices - these are devices that typically transmit data in small portions, often of varying length and usually in a sequential manner;

block devices - these are devices that typically transmit information in large portions, the size of which is usually a multiple of 512 bytes, and allow free access to data;

network devices - devices that perform data transmission on the network.

Character and block devices are provided by the system kernel to the user's space using special files called device files, which are usually located in the `/dev` directory. Each such file, except for its name, has three attributes whose values can be checked by running the `ls -la` command on such file. Files related to character devices are marked with the letter `c`, and files associated with block devices have the label `b`. In addition, each such file has two natural numbers associated with it, called the *major number* and the *minor number*. The major number identifies the driver that supports a specific group of devices, and the minor number identifies a specific device belonging to that group. Minor numbers may be unique to a single driver, but they repeat between different drivers. Major numbers are unique within one device class (e.g. character devices), but they can repeat between classes.

In the kernel space, support for all three categories of devices is associated with the *Virtual File System - VFS*. Each device access request initiated by a user process or user thread is forwarded to the kernel space via system calls, then associated with the corresponding objects in the VFS subsystem and is subject to further processing depending on the category of device or is directly transferred to the device driver that performs them.

This guide describes creating drivers for the character devices. Chapter 1 is dedicated to describing the typical structure and behavior of a character device driver. Chapter 2 describes the API used to implement character device drivers. Chapter 3 explains the role of the `udev` user process in handling character and block devices, and Chapter 4 lists the sample character device driver. Due to the environment used in the laboratory, this is not a physical device, but a pseudo device, i.e. one whose operation is completely simulated by the software. The instruction ends with a list of tasks to be independently implemented as part of the laboratory.

1. Character device drivers

Character device drivers use two VFS objects: a file object and an i-node object. Each character device driver has to perform two basic tasks, which is to initiate cooperation of the device with the system kernel and to provide file object methods that perform operations on the device that will be run as part of system calls.

In the case of physical devices, initiation can mean, for example, providing power to the device and performing diagnostics. The driver is also responsible for providing and registering the interrupt handling

procedure reported by this device and associated bottom halves mechanisms. In addition, the file object's method code must also handle the situation when operations on the physical device may require waiting for them to complete and they may be performed concurrently. Therefore, it becomes necessary to use appropriate synchronization means, which will also allow to stop waiting for the end of the operation, if the process or thread of the user who initiated it receives a signal. Handling of the pseudo character devices is simpler, and moreover, it is largely based on the same mechanisms as support for physical devices. Its basic elements are:

1. obtaining the major and minor number,
2. initializing and adding a `struct cdev` structure to the kernel,
3. initializing and adding a structures related to the `sysfs` file system into the kernel,
4. initializing and adding the structure of the file object methods into the kernel.

If the device allows concurrent access, this must be included in the driver code. Very often, the creators of drivers define their own structures that contain all information about the device and its status. If the driver is to support or simulate the operation of more such devices, then each of them should have its own such structure. The driver's task is also to determine how many user processes or threads can simultaneously use the device. Its creator should, therefore, take care of concurrency or use measures that exclude this concurrency.

2. Description of character device drivers API

From the point of view of the creator of the character device driver, the most important are three types of structures: `struct cdev` - which defines the structure representing the character device in the system kernel, `struct file` - which defines the structure that describes the attributes of the file object, and `struct file_operations` - which defines pointers for functions that perform operations on file object, or methods associated with this object. The file object represents open files on the system, and the last two structures define its class. The character device driver can also use a fourth structure - the `struct inode` structure, which defines the attributes of the i-node object.

Listing 1 presents the definition of the `cdev` structure type. Most of the fields contained in this structure are initialized using the appropriate functions and macros, which will be described later in the guide. The `dev` field contains the device number, which consists of a major and minor number. The `ops` field is a field containing a pointer to the file object's method structure. The `kobj` field is a kernel object associated with the device model and the `sysfs` file system, which is described in the fourth laboratory instruction. The programmer writing the kernel module must remember to directly initialize the `owner` field, which is a pointer to the structure representing the module in which the `struct cdev` type has been declared.

Listing 1: The `struct cdev` definition

```
1 struct cdev {
2         struct kobject kobj;
3         struct module *owner;
4         const struct file_operations *ops;
5         struct list_head list;
6         dev_t dev;
7         unsigned int count;
8 };
```

Listing 2 provides a `struct file` definition that is passed through VFS to most functions pointed to by `struct file_operations` fields. Driver developers most often use the `private_data` pointer field of this structure. As the name suggests, it may point to a memory area containing local (private) driver data. If this area is allocated dynamically, be sure to release it before removing the driver from the system kernel. Other fields, such as `f_op` - a pointer to the file object's method structure, `f_flags` - file open flags, `f_mode` - file access mode, or `f_pos` - file pointer, can also be useful.

Listing 2: The `struct file` definition

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
        struct inode            *f_inode;       /* cached value */
        const struct file_operations    *f_op;

        /*
         * Protects f_ep_links, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to this file */
        struct list_head        f_ep_links;
        struct list_head        f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
} __attribute__((aligned(4)));  /* lest something weird decides that 2 is OK */
```

The definition of `struct file_operations` structure type is provided in Listing 3. Its fields are pointers to functions, which are methods of the file object. They are run from the level of system calls and they directly handle the operation of the device. The creator of the driver does not have to define them all. The most important of them are `open()`, `release()`, `read()` and `write()`. Some character devices offer free access to data. In this case the `llseek()` function is also defined. The way of defining these methods will be described in more detail later in the document. The definitions of the other methods are less common in character device drivers, yet they are also worth to mention. The `unlocked_ioctl()` and `compat_ioctl()` - those methods run from the level of the `ioctl()` call that allow to perform operations on the device that cannot be directly implemented using the other methods. The `poll()` and `fasync()` - those are the methods that notify user processes/threads of the occurrence of new data on the character device.

Listing 3: The `struct file_operations` definition

```
struct file_operations {
         struct module *owner;
         loff_t (*llseek) (struct file *, loff_t, int);
         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 (*read_iter) (struct kiocb *, struct iov_iter *);
```

```
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long,
               unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
        unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,
        unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
 #ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
 #endif
};
```

Not all methods that the `struct file_operations` can point to need to be implemented. In extreme cases, it's enough to implement one of them, e.g. the `read()` method. The most frequently defined methods must meet the following requirements:

`int (*open) (struct inode *, struct file *)` - the method pointed by this field is run from the level of the `open()` system call, i.e. each time the device file is opened from the user space. It may or may not use structures whose addresses are passed to it as call arguments. In the case of the latter variant, most often this method uses the `private_data` field of a `struct file` structure whose address is given by the second argument. The role of this field has been explained before. It is in this method that memory is allocated, whose address is stored in mentioned field. The memory area indicated by it can be used as a convenient point of information exchange between the other methods defined in the driver. In addition, this method performs all work initiating cooperation between the software and a hardware device or pseudo device. If it succeeds, it should return `0`.

`int (*release) (struct inode *, struct file *)` - the method indicated by this field is run from the `close()` system call, i.e. when the device file is closed from the user space. Similar to the method pointed to by the `open` pointer, it may or may not use the call arguments passed to it. Most often, this method carries out activities related to the finalization of the device operation, e.g. turning off power supply or releasing the memory pointed by the `private_data` field of a `struct file` type. Successful completion should be signaled by this function by returning a value of `0` while the exceptions are usually signaled by negative values.

`ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)` - the method indicated to by this pointer is run from the level of the `read()` system call and performs data reading from the device. Through the first call argument the `struct file` type address is passed to this method. The second argument is a pointer to the buffer in the user space in which the method should store the read data. The third argument contains the size of the data requested by the user space, and the fourth argument indicates the variable, that is the file pointer. The last argument is often used to determine which data is to be read and is also used to check that the request does not

exceed the size of the supported device. The first argument is also used to obtain the pointer for private data. The second argument should be handled via the `copy_to_user()` function, because this function also checks the correctness of a given pointer. This method should return the amount of data read, most often expressed by number of bytes. When the read operation ends the function should return `0`. Exceptions are signaled by the following example values: `-EFAULT` - incorrect buffer pointer in user space, `-EIO` - general input-output error, `-EINTR` - reading interrupted by signal.

`ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)` - this pointer indicates the method run from the `write()` system call level and writes data to the device. The role of its call arguments is the same as for the method indicated by the `read` pointer. Only the meaning of the buffer pointed by the second argument changes, from the output buffer to the input buffer, as it now contains data to be written to a device. It should be also handled using the `copy_from_user()` function. The method should return the same values as the one pointed by the `read`, keeping in mind that the returned value now concerns the saved data, not the read one.

`loff_t (*llseek) (struct file *, loff_t, int)` - the function indicated by this pointer is run from the level of the `lseek()` system call and its task is to change the value of the file pointer. Through its first call argument the file object address is given. Through the second argument a new file pointer value is passed, and through the third argument one of three following constant value are given: `SEEK_SET` - it causes that the new value of the file pointer is calculated relative to its beginning, which means that it should be set to the value that was passed to the method by the second argument, after checking its correctness, `SEEK_CUR` - the new value of the file pointer is calculated relative to its current value, i.e. it is set to the sum of its current value and the value given by the second argument of the method, also verifying that the sum is correct, `SEEK_END` - the new file pointer value is calculated relative to the end of the file, i.e. the value passed by the second argument is subtracted from the maximum pointer value for the file and if this difference is correct, the file pointer is set to this value. The method returns a new file pointer value, or `-EINVAL` if the new value was not valid.

Listing 4 presents the `struct inode` structure that together with a method structure, i.e. a `struct inode_operations` structure defines an i-node object class. This second type of method structure will not be described in this document, because it goes beyond its subject. Character device drivers typically only use the `struct inode` structure, and more specifically its `i_rdev` and `i_cdev` fields. The first is a field of type `dev_t` and it contains the device number. Because drivers are most often interested not in the whole number, but in the major number and minor number, this field is not read directly, but with the help of functions that will be described later. The second mentioned field contains a pointer to an `i_cdev` type structure and can be used e.g. by the `open()` or `release()` method to determine which devices supported by the driver must be served in a given call.

Listing 4: The `struct inode` definition

```
truct inode {
        umode_t                 i_mode;
        unsigned short          i_opflags;
        kuid_t                  i_uid;
        kgid_t                  i_gid;
        unsigned int            i_flags;

#ifdef CONFIG_FS_POSIX_ACL
        struct posix_acl        *i_acl;
        struct posix_acl        *i_default_acl;
#endif

        const struct inode_operations   *i_op;
        struct super_block      *i_sb;
        struct address_space    *i_mapping;

#ifdef CONFIG_SECURITY
```

```
        void                    *i_security;
#endif

        /* Stat data, not accessed from path walking */
        unsigned long           i_ino;
        /*
         * Filesystems may only read i_nlink directly.  They shall use the
         * following functions for modification:
         *
         *    (set|clear|inc|drop)_nlink
         *    inode_(inc|dec)_link_count
         */
        union {
                const unsigned int i_nlink;
                unsigned int __i_nlink;
        };
        dev_t                   i_rdev;
        loff_t                  i_size;
        struct timespec         i_atime;
        struct timespec         i_mtime;
        struct timespec         i_ctime;
        spinlock_t              i_lock; /* i_blocks, i_bytes, maybe i_size */
        unsigned short          i_bytes;
        unsigned int            i_blkbits;
        blkcnt_t                i_blocks;

#ifdef __NEED_I_SIZE_ORDERED
        seqcount_t              i_size_seqcount;
#endif

        /* Misc */
        unsigned long           i_state;
        struct mutex            i_mutex;

        unsigned long           dirtied_when;   /* jiffies of first dirtying */
        unsigned long           dirtied_time_when;

        struct hlist_node       i_hash;
        struct list_head        i_io_list;      /* backing dev IO list */
#ifdef CONFIG_CGROUP_WRITEBACK
        struct bdi_writeback    *i_wb;          /* the associated cgroup wb */

        /* foreign inode detection, see wbc_detach_inode() */
        int                     i_wb_frn_winner;
        u16                     i_wb_frn_avg_time;
        u16                     i_wb_frn_history;
#endif
        struct list_head        i_lru;          /* inode LRU list */
        struct list_head        i_sb_list;
        union {
                struct hlist_head       i_dentry;
                struct rcu_head         i_rcu;
        };
        u64                     i_version;
        atomic_t                i_count;
        atomic_t                i_dio_count;
        atomic_t                i_writecount;
#ifdef CONFIG_IMA
        atomic_t                i_readcount; /* struct files open RO */
#endif
```

```
        const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
        struct file_lock_context        *i_flctx;
        struct address_space    i_data;
        struct list_head        i_devices;
        union {
                struct pipe_inode_info  *i_pipe;
                struct block_device     *i_bdev;
                struct cdev             *i_cdev;
                char                    *i_link;
        };

        __u32                   i_generation;

#ifdef CONFIG_FSNOTIFY
        __u32                   i_fsnotify_mask; /* all events this inode cares about */
        struct hlist_head       i_fsnotify_marks;
#endif

        void                    *i_private; /* fs or device private pointer */
};
```

In addition to the structure types described earlier, which are defined in the `linux/fs.h` and `linux/cdev.h` header files, character device drivers also use macros and functions that are available after including the above files and `linux/device.h` header file in the code. Below are descriptions of the most important of them.

`MKDEV(ma,mi)` - a macro that creates and returns the device number (value of type `dev_t` based on the numbers provided to it: major (`ma`) and minor (`mi`)).

`MAJOR(dev)` - a macro that from the given device number reads and returns the major number.

`MINOR(dev)` - a macro that from the given device number reads and returns the minor number.

`int register_chrdev_region(dev_t, unsigned, const char *)` - a function that reserves a specific range of device numbers for the needs of the driver. As the first call argument it takes the first number in the range to be reserved, and as the second argument it takes the amount of these numbers. As the third argument the function takes a pointer to the string being the name of the device. The first device number is usually created using the `MKDEV` macro, specifying the major number and using `0` as the minor number. So the actual challenge is to find the first free major number. If the driver is to be used only on a single computer, then the currently already reserved major numbers can be checked in the `/proc/devices` file. However, if it is to be made available for public use, it is required to contact *The Linux Assigned Names And Numbers Authority* (shortly: `LANANA`) at `http://www.lanana.org/` with a request to assign such a number. Because this process is troublesome, drivers more often use the function described as next. The `register_chardev_region()` function returns zero if the reservation succeeds, or a negative number otherwise.

`int alloc_chrdev_region(dev_t *, unsigned, unsigned, const char *)` - the function assign the specified range of device numbers. The first device number in this range is stored in a variable of type `dev_t`, whose address is passed to the function through its first call argument. The first minor number (usually `0`) is passed through the second argument. The third argument is used to pass the amount of device numbers from the desired range, and the fourth argument is a pointer to the character string being the device name. The function returns zero if the assignment succeeds, or a negative number otherwise.

`void unregister_chrdev_region(dev_t, unsigned)` - a function that unregisters a range of device numbers allocated or reserved by the functions described above. It returns no value, and takes

two call arguments. The first argument is the first device number in the assigned range, and the second argument is the amount of numbers in this range.

`void cdev_init(struct cdev *, const struct file_operations *)` - a function that initializes a structure of type `struct cdev`. The address of this structure is passed as the first call argument, and the `struct file_operations` structure address (the structure of the file object methods) is passed as the second call argument. This function returns nothing.

`int cdev_add(struct cdev *, dev_t, unsigned)` - a function that adds a structure of type `struct cdev` to the system. The address of this structure is passed to the described function as the first call argument. This structure represents a character device in the system kernel and is attached to a list that collects the structures of all such devices. A single structure is created for each device supported by the driver separately. The second call argument is the first device number assigned to the device, and the third argument is the amount of consecutive numbers that are associated with the device. The function returns zero if it succeeds or a negative number otherwise.

`void cdev_del(struct cdev *)` - function that removes a structure of type `struct cdev` from the list of all such structures in the system. It does not return anything.

`class_create(owner, name)` - a macro that creates directories and files related to the driver in the `/sys` directory and it also creates a structure of type `struct class` whose address is then being returned. As the call arguments the function takes respectively: the address of the structure representing the module in the system and the string being the name of the device.

`void class_destroy(struct class *cls)` - a function that releases the memory for the structure created by the `class_create` macro. The address of this structure is passed as its call argument.

`struct device *device_create(struct class *cls, struct device *parent, dev_t devt, void *drvdata, const char *fmt, …)` - a function that creates and returns a structure of type `struct device`, and also sends messages to the `udevd` daemon about a need to create device files. As the first call argument it takes a pointer to a structure of type `struct class`. The second argument is the address of a structure of type `struct device` that will be the parent structure of the newly created structure. If this is not the case then this argument is `NULL`. The third argument is the device number. The fourth argument is a pointer to the data for callback functions. Most often its value is also `NULL`. The fifth argument is a string containing the device name. It may contain formatting strings, so this argument may optionally be followed by other arguments in the amount and types specified by the number and type of the formatting strings.

`void device_destroy(struct class *cls, dev_t devt)` - a function that removes a structure of type `struct device` and sends messages to the `udevd` daemon from the user space to remove the appropriate device files. It takes two call arguments. The first is a structure of type `struct class` and the second is the device number.

`unsigned imajor(const struct inode *inode)` - an `inline` function that reads the major number from the `i_rdev` field of the i-node object and returns it. It takes the i-node object address as its call argument.

`unsigned iminor(const struct inode *inode)` - an `inline` function that reads a minor number from the `i_rdev` field of the i-node object and returns it. It takes the i-node object address as its call argument.

3. The Udev configuration

The description of the `device_create()` and `device_destroy()` functions mentions that they send messages to the user space. These messages are processed by a system called `udev`. Its most important element is the `udevd` daemon[1], which responds to these messages by creating or deleting device files. By default, these files are created in the `/dev` directory and only the `root` user can access them. However, it is possible to provide a specification for this deamon on what attributes such a file should have after creating it. This can be done by creating a file with the appropriate configuration rules and placing it in the `/etc/udev/rules.d/` directory. Listing 5 contains a set of such rules for a pseudo character device created by a driver whose source code contains Listing 6. The number being a prefix of this file's name determines the order in which `udevd` will process this file relative to other rule files. The `KERNEL` token specifies the name of the message to which further tokens apply. The `NAME` token specifies the name of the device file. The `OWNER` token specifies the name of the device file owner. Because the driver was written for Linux Raspbian, the owner will be `pi`. The `MODE` token determines the mode of access to the device file and in this case it is read and write for the owner. There are also other tokens that were not used in this file. Their description, as well as a description of creating rules can be found in the system manual available after using the `man udev` command.

Listing 5: Configuration file `41-fibdev.rules` for the `udevd` daemon

```
KERNEL=="fibdev", NAME="fibdev", OWNER="pi", MODE="0660"
```

4. Example

Listing 6 contains the source code of a pseudo character device that generates consecutive elements of the Fibonacci sequence until their value exceeds the upper limit of the `uint64_t` range. This type is a type introduced into the C99 standard. It allows to store natural numbers and its size of 64 bits is independent of the hardware platform. The elements of this sequence can be displayed on the screen, e.g. using the `cat /dev/fibdev` command.

Listing 6: Driver for the pseudo character device

```
#include<linux/module.h>
#include<linux/fs.h>
#include<linux/cdev.h>
#include<linux/device.h>
#include<linux/uaccess.h>

#define NAME "fibdev"

static uint64_t first, second;

static ssize_t fib_read(struct file *f, char __user *u, size_t size, loff_t* pos)
{
        uint64_t tmp;
        char fibnum[100];
        size_t trans_unit = snprintf(fibnum,sizeof(fibnum),"%llu\n",first);
        if(trans_unit<0)
                return -EIO;
        if(copy_to_user(u,(void *)fibnum,trans_unit))
                return -EIO;

        tmp = first+second;
        if(tmp>=second) {
                first = second;
```

[1] That's the way to call a server-process in Unix

```
                second = tmp;
        } else
                return 0;

        return trans_unit;
}

static ssize_t fib_write(struct file *f, const char __user *u, size_t size, loff_t* pos)
{
        return 0;
}

static int fib_open(struct inode *ind, struct file *f)
{
        first = 0;
        second = 1;
        return 0;
}

static int fib_release(struct inode *ind, struct file *f)
{
        return 0;
}

static struct file_operations fibop =
{
        .owner = THIS_MODULE,
        .open = fib_open,
        .release = fib_release,
        .read = fib_read,
        .write = fib_write,
};

static dev_t number = 0;
static struct cdev fib_cdev;
static struct class *fib_class;
static struct device *fib_device;

static int __init fibchar_init(void)
{
        if(alloc_chrdev_region(&number,0,1,NAME)<0) {
                printk(KERN_ALERT "[fibdev]: Region allocation error!\n");
                return -1;
        }

        fib_class = class_create(THIS_MODULE,NAME);
        if(IS_ERR(fib_class)) {
                printk(KERN_ALERT "[fibdev]: Error creating class: %ld!\n",PTR_ERR(fib_class));
                unregister_chrdev_region(number,1);
                return -1;
        }

        cdev_init(&fib_cdev,&fibop);
        fib_cdev.owner = THIS_MODULE;

        if(cdev_add(&fib_cdev,number,1)) {
                printk(KERN_ALERT "[fibdev]: Error adding cdev!\n");
                class_destroy(fib_class);
                unregister_chrdev_region(number,1);
                return -1;
```

```
84          }
85
86          fib_device = device_create(fib_class, NULL, number, NULL, NAME);
87          if(IS_ERR(fib_device)) {
88                  printk(KERN_ALERT "[fibdev]: Error creating device: %ld!\n",PTR_ERR(fib_device));
89                  cdev_del(&fib_cdev);
90                  class_destroy(fib_class);
91                  unregister_chrdev_region(number,1);
92                  return -1;
93          }
94
95          return 0;
96  }
97
98  static void __exit fibchar_exit(void)
99  {
100         if(fib_device)
101                 device_destroy(fib_class,number);
102         cdev_del(&fib_cdev);
103         if(fib_class)
104                 class_destroy(fib_class);
105         if(number>=0)
106                 unregister_chrdev_region(number,1);
107 }
108
109 module_init(fibchar_init);
110 module_exit(fibchar_exit);
111 MODULE_LICENSE("GPL");
112 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
113 MODULE_DESCRIPTION("A pseudo character device that generates Fibonacci numbers");
114 MODULE_VERSION("1.0");
```

In line 5 of the source code from Listing 6, a header file is included containing the declaration of the `copy_to_user()` function, which was described in the instructions about the `procfs` and `sysfs` file systems. Line 7 defines a string that is the name of the device, which will also be the device's file name. Line 9 defines two variables that will be used to generate the next elements of the Fibonacci sequence. Lines 11-29 contain the `fib_read()` function definition, which is the implementation of the `read()` method of the file object. Lines 13-15 contain local variable declarations. The first of them (`tmp`) has a helper role in generating subsequent elements of the Fibonacci sequence. The second variable `fibnum` is an array of characters in which the specified element of the sequence will be written as a string. The conversion of this element, which is stored in the variable `first`, is done by the function `snprintf()` in line 15. The result of its operation, i.e. the length of the string is stored in the `trans_unit` variable. If this conversion fails, the `snprintf()` function will return a negative value and the `fib_read()` function will terminate by signaling an error with the returned `-EIO` value. This function will behave similarly if copying of the string to the buffer in the user space by the `copy_to_user()` function fails (line 18). In line 21 the value of the next element in the Fibonacci sequence is determined, based on the value of the two previous elements. If it is greater-than or equal to the value of the second of them, then it has been calculated correctly and can be made available to the user space. If not, it means that the upper limit of the `uint64_t` type range has been exceeded. In this case, the Fibonacci string generation should be terminated and the method will return `0`. Please note that the last element in the Fibonacci sequence stored in the `second` variable is not provided to the user space.

The `fib_write()` function, defined in lines 31-34, is an implementation of the `write()` method of the file object, but it does not perform additional actions except for returning `0`. It has been defined only so that the attempt to write to the device does not end in error.

The `fib_open()` function defined in lines 36-41 is an implementation of the `open()` method of the file object. It sets initial values for variables `first` and `second`, and returns `0`.

The `fib_release()` function defined in lines 43-46 implements the `release()` method of the file object. It returns `0` and does not perform any additional actions. Thus, closing a device file, or more

precisely its most recent closing, always succeeds.

Lines 48-55 define the `fibop` structure of the `struct file_operations` type. The pointer fields of this structure are then initialized, to indicate implementations of the file object methods. The `owner` field is assigned with the address of the `struct module` structure returned by the `THIS_MODULE` macro.

In line 57, a variable is declared and initialized that will store the device number. In line 58, a `struct cdev` structure is declared, and lines 59 and 60 contain pointer declarations for `struct class` and `struct device` types. Usually the variables listed in this section are defined as fields of a separate structure, but for a simple character device driver there is no such need.

Lines 62-96 contain the definition of the module constructor. In line 64, a device number is assigned to the driver. Due to the fact that it will support only one pseudo device, you only need one such number. If this assignment fails, an appropriate message will be put in the kernel buffer (line 65) and the function will end by returning the value `-1`. In line 69, a `struct class` structure is created, as well as the corresponding directories and files in the `sysfs` file system. Line 70 checks whether the latter operation was successful. If not, the error code will be contained in the returned address. In this case, the constructor writes this code to the kernel buffer, then (line 72) releases the assigned device number and (line 73) exits. On line 76, a `struct cdev` structure is initialized. Its `owner` field must be initialized separately (line 77) with the address returned by the `THIS_MODULE` macro. In line 79, this structure is added to the system. If this operation fails, then in addition to placing the appropriate message in the kernel buffer, the constructor releases the previously created structure and device number, and then exits. On line 86, a `STRUCT DEVICE` structure is created and a message is sent to the `udevd` daemon from user space. If this operation fails, the effects of all previous operations are rolled back before the constructor finishes signaling the error. However, if this operation is successful, the constructor exits with zero.

The module destructor is defined in lines 98-107. In this function, memory for `struct device` and `struct class` (lines 101 and 104) is released. The `cdev` structure (line 102) is removed from the system and the device number (line 106) is released, provided that operations related to these elements were successful in the constructor. Please note that these operations are performed in reverse order to their constructor counterparts. This is the correct order of finalization.

Exercises

1. `[3 points]` Change the constructor code from Listing 6 so that it uses the `goto` instruction to handle exceptions, but behaves the same.

2. `[5 points]` Change the module code from Listing 6 so that the elements of the Fibonacci sequence are written to the array and implement the `llseek()` method so that you can indicate the element that should be read. Write program for the user space that will use this driver property.

3. `[7 points]` Write a driver that will support two pseudo character devices. The first device will return the next natural number in relation to the number written to it, and the second device will return the previous natural number in relation to the number written to it. These input numbers should fall within the `u64` range. Remember to synchronize read and write.

4. `[3 points]` Change the module code from Listing 6 so that the `fib_open()` function returns only `0`, but that the module operation is preserved.

5. `[5 points]` Write a kernel module that will create a pseudo character device called `clipboard` that allows you to write (e.g. using the `echo` command) and read a string not longer than `1024` characters. Remember to synchronize read and write.

6. `[7 points]` Write a driver of a pseudo character device that will return the text provided to it through a file in the `sysfs` file system. Remember to synchronize the reading and writing from the `sysfs` file system level.