Laboratory 4: „The `procfs` and `sysfs` file systems” (two weeks)

dr inż. Arkadiusz Chrobot
dr inż. Karol Tomaszewski

April 5, 2024

Contents

Introduction

The first laboratory guide presents the use of parameters for kernel modules. They allow one-direction and one-time communication with the kernel module, i.e. when loading it, the user is able to determine its behavior using parameters. However, the Linux kernel provides other mechanisms that allow two-way communication with modules, which can additionally take place during their execution. These include virtual file systems such as `procfs` and `sysfs`, which will be the subject of this manual. Chapter 1 describes the `procfs` file system, its role in the system, usage, API, and an example module using this file system. Chapter 2 contains similar information about the `sysfs` file system. The instruction ends with a list of tasks to be carried out within the laboratory.

1. The `procfs` file system

The `procfs` file system exists only in the computer's operating memory, which means that its directory structure and file contents are generated while the kernel is running. Mainly it is statistical information on the operation of the hardware, individual kernel subsystems and user processes. They are placed in text files, so they can be displayed on the screen, e.g. using the `cat` shell command. For example, information about the processor or processors installed on computer can be obtained by issuing the `cat /proc/cpuinfo` command. There are also command-programs that interpret and display information in `procfs` files in a more compact form. The `pmap` command is an example of such command. Some of these files are also modifiable and are used for dynamic configuration (i.e. during the execution) of the operating parameters of the kernel subsystems. Most often, such files can be modified by the redirection of the `echo` command output with the use of the redirection operator: `>`. On most Linux distributions, the `procfs` file system is mounted in the `/proc` directory.

1.1. API description

The handling of `procfs` files has changed in version 3.10 of the kernel. These changes are described at `http://tuxthink.blogspot.com/2013/10/creating-read-write-proc-entry-in.html`.

Most of the functions and data structures related to this file system are contained in the `linux/proc_fs.h` header file. The main structure describing files and directories in `procfs` is struct `struct proc_dir_entry`. Unfortunately, starting from the mentioned version of the kernel it is not available for developers of the kernel modules directly. The main kernel developers have decided to hide the details of its implementation to prevent many dangerous mistakes made by the module developers. The description of functions related to handling the `procfs` will start with those that allow to create or delete files and subdirectories. Here are some of them:

`struct proc_dir_entry *proc_mkdir(const char *, struct proc_dir_entry *)` - this function creates a new subdirectory in the directory where the `procfs` system is mounted. Its first call argument should be the name of the directory to be created. The second argument is a pointer to of type `struct proc_dir_entry` associated with the parent directory for the one to be created. If the value of this argument is `NULL`, a subdirectory will be created in the main directory of the `procfs` file system. The function returns `NULL` if it fails or it returns a pointer of type `STRUCT PROC_ENTRY` of the newly created subdirectory.

`struct proc_dir_entry *proc_mkdir_mode(const char *, umode_t, struct proc_dir_entry *)` - this function works similarly to `proc_mkdir()`, but it takes an additional (second) call argument that specifies the access rights to the created directory. These rights can be stored in the form of octal number or as a bit sum of the following constant variables:

`S_IRWXU` - all access rights for the file/directory owner,

`S_IRUSR` - read access for file/directory owner,

`S_IWUSR` - write access for the owner,

`S_IXUSR` - right to execute for the owner,

`S_IRWXG` - all access rights for the group to which the file/directory owner belongs,

`S_IRGRP` - read access for the group to which the file/directory owner belongs,

`S_IWGRP` - write access for the group to which the file/directory owner belongs,

`S_IXGRP` - right to execute for the group to which the file/directory owner belongs,

`S_IRWXO` - all access rights for other users,

`S_IROTH` - read access for other users,

`S_IWOTH` - write access for other users,

`S_IXOTH` - right to execute for other users,

`S_IRWXUGO` - all access rights for all users,

`S_IRUGO` - read access for all users,

`S_IWUGO` - write access for all users,

`S_IXUGO` - right to execute for all users.

Those variables can be also used one at a time.

`struct proc_dir_entry *proc_create(const char *name, umode_t mode, struct proc_dir_entry *parent, const struct file_operations *proc_fops)` - this function creates a file in the `procfs` file system. As the first call argument, it takes the name of the file to be created, and as the second argument the function takes access rights, which are explained in the description of the `proc_mkdir()` function. The next call argument is a pointer of type `struct proc_dir_entry` associated with the directory where the file is to be created. The last call argument of this function is a pointer to a `struct file_operations` structure, which contains pointers to functions that perform operations on the file (will be explained later in this chapter). The `proc_create()` function returns a pointer to a `proc_dir_entry` structure associated with the newly created file, or `NULL` if the file could not be created.

`struct proc_dir_entry *proc_create_data(const char *, umode_t, struct proc_dir_entry *, const struct file_operations *, void *)` - this function is similar to `proc_create()`, but takes an additional, last, argument as a pointer (type `void *`) on buffer, where the data contained in the file will be stored.

`struct proc_dir_entry *proc_symlink(const char *, struct proc_dir_entry *, const char *)` - this function is used to create a symbolic link to a file belonging to the `procfs` file system. It takes three call arguments: the name of the link, a pointer to a `proc_dir_entry` structure that describes the directory in which the link is to be created, and the name of the file to which the link should be created. The function returns a pointer to a `struct proc_dir_entry` associated with the newly created link, or `NULL` if the link operation was unsuccessful.

`void proc_remove(struct proc_dir_entry *)` - this function is used to delete a file or directory from the `procfs` file system. It returns no value and as a call argument it takes the pointer to a `struct proc_dir_entry` associated with the file or directory to be deleted.

Modules using `procfs` use the `struct file_operations` structure, which allows to define functions (methods) that perform basic file operations, such as read and write. Listing 1 shows this structure.

Listing 1: Definition of the `struct file_operations` structure

```
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
 };
```

Even a simple analysis of the definition of this structure shows that it is very complex and allows to define many methods that work on files. Fortunately, kernel programmers don't have to define them all. The most commonly defined are: `open()`, `release()`, `read()`, `write()` and `llseek()` and the `owner` field value is set to prevent other kernel subsystems from removing this structure from memory. To further facilitate the work of module programmers, a *sequential file* mechanism has been defined in the kernel that includes ready-made methods for performing typical file operations that are most commonly used in the `procfs` system. So in some cases, it is enough to define three functions: to handle writing to a file, to handle reading from a file, and for opening a file. The headers of the first two functions must comply with the following prototypes:

```
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

and:

```
            ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Both functions return the number of bytes written/read from the file, or `-1` if an exception occurs. The first call argument of these functions is a pointer to the structure associated with the open file on which the operation is to be performed. The second call argument is a pointer to a buffer related to the user space (hence the `__user` marker before their declaration), from which the write function will retrieve information to be stored in the file, and the read function will place data from the file transferred to the user space. The third call argument is the size of the read/write portion of the file expressed in bytes, and the last argument is a pointer to the variable locating the beginning of this portion of information (file pointer).

Two following functions defined in the `linux/uaccess.h` file will be useful in defining the `write()` and `read()` functions:

`long copy_to_user(void __user *to, const void *from, unsigned long n)` - this function allows to copy data from the buffer located in the kernel space and indicated by the `form` pointer to the buffer located in the user space and indicated by the `to` pointer. The size of the data to be copied is specified (in bytes) by the value of the `n` argument. The function returns the number of bytes that it **could not** copy, so if it succeeds it will return `0`.

`long copy_from_user(void *to, const void __user * from, unsigned long n)` - this function copies the data from the buffer located in user space and indicated by the `from` parameter to the buffer located in kernel space and indicated by the `to` pointer. The `n` argument specifies the number of bytes to copy. The function returns the number of bytes that **could not** be copied, so if it succeeds it will return `0`.

The `owner` field of the `struct file_operations` structure is assigned with the value of `THIS_MODULE` macro. It is available in every module and returns a pointer to a `struct module` representing the module in the Linux kernel.

1.2. Sequential files

To simplify the creation of simple file systems whose files are mainly read sequentially, kernel developers have created the appropriate functions and structures that together form a mechanism simply called *sequential files*. It requires to define of several functions whose task is to enable easy iteration over the contents of the file, even if its size exceeds the size of the page. On the other hand, this mechanism also provides ready-made functions that can act as methods for file objects. Contrary to the name, the described solution also allows free access to files, but it is not very effective. The sequential file mechanism can be successfully applied to kernel modules using `procfs` file system[1].

1.2.1. Description of the sequential file API

Functions and data structures related to the sequence file mechanism are declared or defined in the `linux/seq_file.h header file`. One of the basic structures of this mechanism is the `struct seq_operations`, whose structure is shown in Listing 2.

Listing 2: Definition of the `struct seq_operations` structure

```c
struct seq_operations {
        void * (*start) (struct seq_file *m, loff_t *pos);
        void (*stop) (struct seq_file *m, void *v);
        void * (*next) (struct seq_file *m, void *v, loff_t *pos);
        int (*show) (struct seq_file *m, void *v);
};
```

This definition shows that to use the sequential file mechanism it is needed to define four functions to create an iterator that access the file contents sequentially. The addresses of these functions should then be assigned to the appropriate pointers under the structure fields from Listing 2.

[1]The mechanism of sequential files has been described, among others, on the web page `http://crashcourse.ca/introduction-linux-kernel-programming/lesson-13-proc-files-and-sequence-files-part-3`.

The `start` pointer should indicate the function that will be responsible for initiating the iterator's operation. It must have two call arguments. First argument is the pointer to a structure representing the sequential file, and the second argument is the address of the file pointer set to the beginning of the file[2]. The function should return the address of the first buffer element to read from the sequential file.

The `stop` pointer should indicate the function that finalizes the iterator. This function should have two call arguments. The first is a pointer to a structure representing a sequential file, and the second argument is a pointer to the buffer containing the data placed in the sequential file. If this buffer was created by dynamic memory allocation in the function indicated by the `start`, then this buffer should be removed by releasing the memory in the function indicated by the `stop`. This function returns nothing. In many cases, its definition remains empty if there is no action to perform to finalize the iterator.

The `next` pointer should indicate a function that will return the pointer to the current element to be read from the sequential file and set the file pointer to the next element afterwards. If such an element does not exist, it should return `NULL`. This function must have three call arguments. The first call argument points to the structure representing the sequential file, the second argument is the pointer to the buffer, and the third argument is the address of the file pointer. Before the function finishes, the file pointer should be updated to point to the next element to be read from the buffer.

The `show` pointer should point to a function that will put data from the buffer into the file. It should have two call arguments. The first call argument is the address of the structure representing the sequential file, and the second argument should indicate the data to be transferred to the user space. It returns zero if the read succeeds. The following functions may be useful in its implementation:

`void seq_printf(struct seq_file *m, const char *fmt, …)` - this function has a similar list of call arguments and usage as `printf()` from user space. Its first call argument is a pointer to a structure representing a sequential file, followed by a character string containing only the output string or formatting strings. The next are variables (buffer elements) whose values are to be substituted into formatting strings.

`void seq_putc(struct seq_file *m, char c)` - this function allows to read a single character from the sequential file buffer. As the first call argument it takes the address of the structure representing the sequential file, and as the second argument it takes a pointer to a character that will be read from the sequential file.

`void seq_puts(struct seq_file *m, const char *s)` - this function allows to read a string from the sequential file buffer. As the first call argument it takes the address of the structure representing the sequential file, and as the second argument the string address that is in the buffer of the sequential file and is to be read from it.

At the beginning of the section 1.2 on sequential files, there is information that their mechanism contains ready-made functions that can be used as methods for a file object. Those functions were also declared in the `linux/seq_file.h` header file. They consist of:

`int seq_open(struct file *, const struct seq_operations *)` - this function opens the sequential file, i.e. initiates work with it. It is most often called inside of the definition of function (*wrapped by the function*), which will be pointed to by the `open` pointer of a `struct file_operations` structure. As the first call argument, the second argument of the *wrapping function* is passed, and the second function argument is the address of the initialized `struct seq_operations` structure. It returns the opening code, which should also be returned by the function implementing the `open()` method.

`int single_open(struct file *, int (*)(struct seq_file *, void *), void *)` - this function is a replacement for `seq_open()` if a single value is to be read from the file. As the first call argument, the last argument of the function implementing the `open()` method is passed, and the second argument is the pointer to the `show()` function for the sequential file. As the last call argument the function takes the address of a buffer for the sequential file.

`int single_release(struct inode *, struct file *)` - the address of this function is assigned to the `release` field of a `struct file_operations` structure if the `single_open()` function was used to implement the `open()` method.

[2]For this function the initial value of the file pointer does not necessarily have to be zero.

`int seq_release(struct inode *, struct file *)` - the address of this function is assigned to the `release` field of a `struct file_operations` structure if the `seq_open()` function was used to implement the `open()` method.

`ssize_t seq_read(struct file *, char __user *, size_t, loff_t *)` - the address of this function is assigned to the `read` field of a `struct file_operations` structure.

`loff_t seq_lseek(struct file *, loff_t, int)` - the address of this function is assigned to the `llseek` field of a `struct file_operations` structure.

1.3. Example

Listing 3 shows the source code for the kernel module, which creates the `procfs_test` subdirectory in the `/proc` directory, and contains a file called `procfs_file` that can be written and read from user space.

Listing 3: Module using the `procfs` file system

```c
#include<linux/module.h>
#include<linux/uaccess.h>
#include<linux/proc_fs.h>
#include<linux/seq_file.h>

static struct proc_dir_entry *directory_entry_pointer, *file_entry_pointer;
static char *directory_name = "procfs_test", *file_name = "procfs_file";
static char file_buffer[PAGE_SIZE];

static int procfsmod_show(struct seq_file *seq, void *data)
{
        char *notice = (char *)data;
        seq_putc(seq,*notice);
        return 0;
}

static void *procfsmod_seq_start(struct seq_file *s, loff_t *position)
{
        loff_t buffer_index = *position;
        return (buffer_index<PAGE_SIZE && file_buffer[buffer_index]!='\0')
                ?(void *)&file_buffer[buffer_index]:NULL;
}

static void *procfsmod_seq_next(struct seq_file *s, void *data, loff_t *position)
{
        loff_t next_element_index = ++*position;
        return (next_element_index<PAGE_SIZE && file_buffer[next_element_index]!='\0')
                ?(void *)&file_buffer[next_element_index]:NULL;
}

static void procfsmod_seq_stop(struct seq_file *s, void *data)
{
}

static struct seq_operations procfsmod_seq_operations = {
        .start = procfsmod_seq_start,
        .next = procfsmod_seq_next,
        .stop = procfsmod_seq_stop,
        .show = procfsmod_show
};

static int procfsmod_open(struct inode *inode, struct file *file)
{
        return seq_open(file, &procfsmod_seq_operations);
}

static ssize_t procfsmod_wirte(struct file *file, const char __user *buffer, size_t count,
                 loff_t *position)
{
        int length = count;
        if(count>PAGE_SIZE)
                length=PAGE_SIZE-1;
        if(copy_from_user(file_buffer,buffer,length))
                return -EFAULT;
        file_buffer[length]='\0';
        return length;
}

static struct file_operations procfsmod_fops = {
        .owner = THIS_MODULE,
        .open = procfsmod_open,
        .read = seq_read,
        .write = procfsmod_wirte,
        .llseek = seq_lseek,
        .release = seq_release
};

static int __init procfsmod_init(void)
{
        directory_entry_pointer = proc_mkdir(directory_name, NULL);
        if(IS_ERR(directory_entry_pointer)) {
                pr_alert("Error creating procfs directory: %s. Error code: %ld\n",
                        directory_name,PTR_ERR(directory_entry_pointer));
                return -1;
        }
        file_entry_pointer = proc_create_data(file_name,0666,directory_entry_pointer,
                                &procfsmod_fops,(void *)file_buffer);
        if(IS_ERR(file_entry_pointer)) {
                pr_alert("Error creating procfs file: %s. Error code: %ld\n",
                        file_name,PTR_ERR(file_entry_pointer));
                proc_remove(directory_entry_pointer);
                return -1;
        }

        return 0;
}

static void __exit procfsmod_exit(void)
{
        if(file_entry_pointer)
                proc_remove(file_entry_pointer);
        if(directory_entry_pointer)
                proc_remove(directory_entry_pointer);
}

module_init(procfsmod_init);
module_exit(procfsmod_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_DESCRIPTION("Procfs capabilities showing module.");
MODULE_VERSION("1.0");
```

Line 6 of Listing 3 contains declaration of two pointers for `struct proc_dir_example` structures.

The first will point to the structure associated with the created subdirectory, and the second will point to the structure associated with the created file. In line 7, two variables were declared and initiated, which point to the strings being the names of the subdirectory and file, respectively, created by the module. The next line contains the declaration of the variable being the buffer for the file data. It is an array of 4KiB characters, i.e. it contains 4096 elements.

The `procfsmod_show()` function (lines 10-15) implements the `show()` function for a sequential file. In the body of this function, a `void *` pointer is cast to the pointer on a character (of type `char*`), and then the character pointed to by this pointer is placed in a file called `procfs_file` using the function `seq_putc()` and the value of `0` is returned.

The `procfsmod_seq_start()` function (lines 17-22) is in an implementation of the `start()` function for a sequential file. In the body of this function, in line 19, a local variable is declared, to which the initial value of the file pointer is assigned. The function then checks (line 20) whether the pointer is larger than the file buffer size or that the item it specifies does not contain the string end character (`\0`). If the above conditions are not met, the function returns the address of the first buffer element to read, otherwise it returns `NULL`.

The `procfsmod_seq_next()` function (lines 24-29) is an implementation of the `next()` function for a sequential file. It works a slightly different than explained in the `next()` description. This function in line 26 first determines the index of the next element in the buffer to be read and stores it to the local variable `next_element_index`. Then it checks the same conditions as `procfsmod_seq_start()`. If, after checking them, it turns out that the index value is correct, then this function returns the address to the buffer element it specifies, and if its not correct, the function returns `NULL`.

The `procfsmod_seq_stop()` function (lines 31-33) is an implementation of the `stop()` function for a sequential file. Its body is empty, because the implementation of the `procfs` file used in the module does not require any operations related to the iterator finalization. The file buffer is created statically in the module and it does not require to release the memory allocated to it.

Lines 35-40 contain a declaration and initialization of the `procfsmod_seq_operations` structure of type `struct seq_operations`. The addresses of the respective functions described earlier are assigned to the respective structure pointer fields.

The `procfsmod_open()` function (lines 42-45) is an implementation of the `open()` method for a file object. In its body the function `seq_open()` is called, which as a call argument takes a file object pointer (the second argument of the `procfsmod_open()` function) and the address of the `procfsmod_seq_operations` structure containing the addresses of functions implementing operations on the sequential file. The code returned by `seq_open()` is also returned by the `procfsmod_open()` function.

The `procfsmod_write()` function (lines 47-57) is an implementation of the `wirte()` method for a file object. In its body, in line 50, the number of bytes to write to the file buffer from user space is stored in the local variable `length`. It is passed to the function by the `count` call argument. In the next line, the function checks if this number is greater than the buffer size. If so, the index of the last buffer element is stored in the variable `length`. This prevents the buffer area from being exceeded. We assume that the maximum file size that the module will use cannot exceed 4 KiB. Then, the data from the buffer belonging to the user space and indicated by the `buffer` pointer is copied to the file buffer using the `copy_from_user()` function. If the copy operation fails, the function will return the appropriate exception number. In line 55, the last element of the file buffer is written with the *end of string* character. This is necessary if the process from the user space attempts to write more information than 4 KiB to the file. In this case, it needs to be ensured that the copied string ends. After successful copying, the function returns the number of bytes copied.

In lines 59-66 the operation structure (methods) for the file object is declared and initiated. Its individual pointer fields are assigned with the addresses of relevant functions defined in the module or ready-made functions, which are provided by the sequential file mechanism.

In the module's initialization function, a subdirectory of the `/proc` directory is first created using the `proc_mkdir()` function. The file name is given as the first call argument. The second call argument is the address of the structure associated with the parent directory of the created directory, therefore its value is given as `NULL`. If the value of this argument is `NULL`, it means that this directory will be the mount point for the `procfs` file system, which is usually the `/proc` directory. After calling the function, the value of the returned address is checked. If an exception occurs, an appropriate message is placed in the kernel buffer, a value indicating the operation error is returned, and the initialization function is terminated. However, if the directory was successfully created, a file called `procfs_test` is created in line 76 using the `proc_create_data()` function. The first argument to call this function is a variable indicating the name of the file being created, then an octal number specifying the access rights to it, the address of the structure describing the directory in which the file will be created, the address of the operation structure (methods) for the file object and finally the buffer address on data for this file[3]. The result of this function is also checked in case an exception occurs during its execution. If this happens, as before, the appropriate message is placed in the kernel buffer, and the created subdirectory is removed with the use of `proc_remove()` function. Only after completing this action does the initialization function exit, returning the value `-1`. However, if the file creation is successful, the initialization function will return `0` and also exit.

After completing the initialization function, the user can save the strings to the `procfs_file`, e.g. in this way:

```
echo "test" > /proc/procfs_test/procfs_file
```

To read the contents of the file, the user can use, for example, the following command:

```
cat /proc/procfs_test/procfs_file
```

The file and directory are removed when the module is removed from the system kernel. This is done by the module cleaning (finalizing) function, which first checks whether the directory and file were successfully created, i.e. whether the pointers on the structures describing them have value other than `NULL`. If so, then each of these pointers is passed as an argument to the `proc_remove()` function call.

2. The sysfs file system

The `sysfs` file system is closely related to the device model, i.e. the Linux kernel subsystem, which was introduced into it to facilitate device management activities such as representing connections topology, creating hierarchy and classification of devices and energy management. The basic element of this model is the kernel object structure. Each kernel object is represented as a directory in the `sysfs` file system, which in most Linux distributions is mounted in the `/sys` directory. Kernel objects represent the topology of device connections that make up a computer system supervised by Linux, so the `sysfs` structure reflects this topology in user space. In addition, each kernel object can have attributes that are visible in the user space as files. The content of these files can be read by most users, but only privileged users (usually the *root*, or users that belong to the same group as the *root*) can modify them. Attribute-related files are generally text files, but there are also binary files among them. Usually a single file contains a single value. Most of these files can be read using the `cat` shell command mentioned above. The modification usually requires access to the system administrator shell. It can be achieved for a properly configured user by using the `sudo su` command. The `sysfs` file system also emits event notifications for user space if the value of any attribute is modified by the kernel subsystem. For example, the `udev` daemon is notified when a new external device is added to the system. This topic will not be described in more detail in this document, because it requires knowledge about communication via sockets using the *netlink* protocol. However this will be the subject of a future laboratory guide.

2.1. API description

The most important data structures in the device model used in Linux are `struct kobject` structures. These structures are simply kernel objects. The class of such objects (the type) is determined by another type of structure, called `kobj_type`. A single such structure can be associated with many kernel objects, thereby giving them the same characteristics (attributes and behavior). The third important type of structure in described model is `struct kset`. These are containers (sets) of kernel objects that perform similar functions, e.g. they are associated with drivers of a certain group of devices.

Listing 4 contains the definition of the kernel object structure type, which can be found in the `linux/kobject.h` header file.

Listing 4: Definition of the `struct kobject` structure

[3]This is, in short, a file buffer.

```
struct kobject {
        const char              *name;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct kernfs_node      *sd; /* sysfs directory entry */
        struct kref             kref;
#ifdef CONFIG_DEBUG_KOBJECT_RELEASE
        struct delayed_work     release;
#endif
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
        unsigned int uevent_suppress:1;
};
```

The `name` field in this structure indicates a string that is the name of the object and thus of the name of the associated directory in the `sysfs` file system. The `parent` field indicates a kernel object that is superior to the one being described. In the `sysfs` file system, this is the parent directory. The `ktype` and `kset` pointer fields store the address of a class of the object and the set, respectively. The `kref` field is used to count the object references. Other fields are not important from the point of view of this guide.

Listing 5 contains a definition of the structure describing the class of objects.

Listing 5: Definition of the `struct kobj_type` structure

```
struct kobj_type {
        void (*release)(struct kobject *kobj);
        const struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;
        const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
        const void *(*namespace)(struct kobject *kobj);
};
```

The `release` pointer field indicates the function that is responsible for finalizing (cleaning) the object when it is no longer used. As the call argument, this function takes the address of the object and is called automatically when such object is destroyed. The `sysfs_ops` pointer field indicates a structure of the `sysfs_ops` type, the definition of which is given in Listing 6. The last field of this structure that will be described in this document is `default_attrs`. It is a pointer to an array of `struct attribute` structures. The definition of this structure type is given in Listing 7. These structures are represented in the `sysfs` file system as files.

Listing 6 contains the definition of the `struct sysfs_ops` type.

Listing 6: Definition of the `struct sysfs_ops` structure

```
struct sysfs_ops {
        ssize_t (*show)(struct kobject *, struct attribute *, char *);
        ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
};
```

This structure contains two pointers to functions that play the role of object methods. The first function is called when a *file associated with the kernel object* is read from user space. The address of the object is passed to it as the first call argument, and the second argument is the address of the attribute, which is represented as the mentioned *file* in the `sysfs` file system. The third call argument is the address of the 4 KiB buffer to which the data from the file needs to be copied. The second function is called when the content of a file is saved from user space. The first three call arguments are similar to those described earlier. The fourth call argument is the number of bytes to be saved in the file. This function have to copy the data from the buffer, with the size specified by the fourth call argument, into the corresponding variable in the kernel module. Both of these functions must be defined by the programmer whose module uses the file on the `sysfs` system.

Listing 7 contains a definition of the `struct attribute` type, which represents files in the `sysfs` system.

Listing 7: Definition of the `struct attribute` structure

```c
struct attribute {
        const char              *name;
        umode_t                 mode;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        bool                    ignore_lockdep:1;
        struct lock_class_key   *key;
        struct lock_class_key   skey;
#endif
};
```

Among the structure fields there are: `name` which is the name of the attribute as well as the name of the file, and `mode`, which specifies the access rights to the file.

If for some reason we do not want to define the kernel object class, but we want to use the files in the `sysfs` file system in the module, we can use the `struct kobj_attribute` structure. This type definition is presented in Listing 8.

Listing 8: Definition of the `struct kobj_attribute` structure

```c
struct kobj_attribute {
        struct attribute attr;
        ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf);
        ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr,
                         const char *buf, size_t count);
};
```

This structure contains a field of type `struct attribute`, which is represented in the `sysfs` system as a file, and two pointer fields for functions that are called when the file is read and written, respectively. Please note that the pointer types of these functions are similar to those of the `struct sysfs_ops` structure. It differs only in the type of the second call argument of the function. Instead of `struct attribute` it is `struct kobj_attribute`.

The following functions[4] are used to handle kernel objects, which become available when the `linux/kobject.h` header file is included in the module code:

`void kobject_init(struct kobject *kobj, struct kobj_type *ktype)` - this function is used to associate the kernel object to which its pointer is passed as its first call argument with the structure specifying the type of this object to which the pointer is passed as its second call argument. The function returns nothing.

`int kobject_add(struct kobject *kobj, struct kobject *parent, const char *fmt, …)` - this function does two things. At first, it gives the kernel object, to which its pointer is passed as its first argument, the name specified by the `fmt` format string. This string is used like in the case of the `printf()` function known from user space. The second operation performed by this function is to add mentioned kernel object to the structure describing the hierarchy of all such objects created in the system kernel. The second call argument of this function points to the object that will be the parent of the object pointed to by the first call argument. Because in the `sysfs` file system kernel objects are represented as directories, the object pointed to by the first argument of the function will then be a subdirectory of the second object. If the second call argument is `NULL`, then two following scenarios are possible. If the object pointed to by the first call argument is assigned to a specific set, then it will be linked to the object of that set. Otherwise, it will be visible as a

[4]Similar to the example module for the `procfs` file system, this guide mainly describes the functions used in the example.

directory in root of the `sysfs` system directory. The function returns a negative number when an exception occurs.

`extern struct kobject * __must_check kobject_create_and_add(const char *name, struct kobject *parent)` - this function creates a kernel object, gives it a name and places it in the hierarchy of kernel objects. As the first call argument it takes the string being the name of the object. As the second argument it takes a pointer to the object that will be its parent in the hierarchy. This argument has the same rules as described in the `kobject_add()` function. The kernel object is created dynamically by this function, i.e. memory is allocated to this variable. If the object creation fails, the function returns `NULL`. Otherwise it returns the address of the newly created object. The `__must_check` tag (macro) tells the compiler to check that the function result is assigned to a variable.

`void kobject_put(struct kobject *kobj)` - this function reduces the reference count for the kernel object to which the pointer is passed to it as a call argument. If this counter reaches 0, this function removes this kernel object.

`void kobject_del(struct kobject *kobj)` - this function removes the kernel object, to which its pointer is passed as a call argument, from the hierarchy of all kernel objects and calls the `kobject_put()` function.

Functions and macros that handle kernel object attributes, visible as files in the `sysfs` file system, become available when the `linux/sysfs.h` header file is included in the module's code. Here are descriptions of some of them:

`__ATTR(_name, _mode, _show, _store)` - this macro initializes a `struct kobj_attribute` structure that that is represented as a file in the `sysfs` file system. The first argument of the macro is the file name, the second argument is an octal number specifying the file's access rights, or an expression consisting of constant variables corresponding to the individual rights. As described earlier in this guide, it is not possible on the `sysfs` file system to grant modification rights to an ordinary user. The last two call arguments are pointers to the functions used to read and write on the file respectively. The implementation of these functions has also been described previously in this guide.

`int __must_check sysfs_create_file(struct kobject *kobj, const struct attribute *attr)` - this function adds the attribute described by the `struct attribute` type pointed by the second call argument to the kernel object indicated by its first call argument. Thus, it creates a file in the `sysfs` file system which is a file located in the directory represented by the mentioned kernel object. If the function fails on adding the attribute to the object, it returns a negative number, which is the exception code.

`void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr)` - this function removes the attribute described by the structure pointed by the second call argument from the kernel object indicated by the first call argument. Thus, it removes the file associated with the attribute from the directory represented by mentioned kernel object.

2.2. Examples

In the first example, with the source code presented in Listing 9, the kernel module creates files in the `sysfs` file system. Processes from the user space can write or read a single integer from this file. Its value is then placed in the module variable, so in this way user processes can pass information to a kernel module as well as receive information from it.

Listing 9: Module using the sysfs file system

```c
#include<linux/module.h>
#include<linux/sysfs.h>
#include<linux/kobject.h>

static struct kobject *kernel_object;
static int number;
```

```
7
8  static ssize_t
9  number_show(struct kobject *kernel_object, struct kobj_attribute *attribute, char *buffer)
10 {
11         return sprintf(buffer,"%d\n",number);
12 }
13
14 static ssize_t
15 number_store(struct kobject *kernel_object, struct kobj_attribute *attribute, const char *buffer,
16                 size_t count)
17 {
18         sscanf(buffer,"%d",&number);
19         return count;
20 }
21
22 static struct kobj_attribute number_kattribute =
23 __ATTR(number,0664,number_show,number_store);
24
25 static struct attribute *number_attribute = &number_kattribute.attr;
26
27 static int __init sysfs_test_init(void)
28 {
29         kernel_object = kobject_create_and_add("test",&THIS_MODULE->mkobj.kobj);
30         if(!kernel_object)
31                 goto err;
32
33         if(sysfs_create_file(kernel_object,number_attribute))
34                 goto err1;
35
36         return 0;
37         err:
38                 printk(KERN_ALERT "Could not create a kobject!\n");
39                 return -ENOMEM;
40         err1:
41                 kobject_del(kernel_object);
42                 printk(KERN_ALERT "Could not create a sysfs file!\n");
43                 return -ENOMEM;
44 }
45
46 static void __exit sysfs_test_exit(void)
47 {
48         sysfs_remove_file(kernel_object,number_attribute);
49         if(kernel_object)
50                 kobject_del(kernel_object);
51 }
52
53 module_init(sysfs_test_init);
54 module_exit(sysfs_test_exit);
55 MODULE_DESCRIPTION("A kernel module demonstrating the usage of sysfs.");
56 MODULE_LICENSE("GPL");
```

Line 5 of the module code contains a pointer to the kernel object that will be created dynamically, and line 6 contains a declaration of the `number` variable, the value of which will be read and written from the user space using the file created by the module in the `sysfs` file system.

Lines 8-12 contain the definition of the `number_show()` function, which is called when a file in the `sysfs` file system is read. In the body of this function (line 12), with the help of the `sprintf()` function, the value placed in the `number` variable is converted into a string and placed in the buffer pointed to by the function argument called `buffer`. The `show_number()` function returns a value of how many bytes it has written to this buffer, i.e. the information returned by `sprintf()` and then exits.

Lines 14-20 contain the definition of the `number_store()` function, which stores a value in the `number` variable that was given from the user space in form of a string in the buffer indicated by its `buffer` argument. The conversion of the string to a numeric value and storing it to the variable is performed using the `sscanf()` function. The `number_store()` function returns the number of bytes read from the mentioned buffer, which is stored in its `count` parameter and then terminates.

The attribute structure of type `struct kobj_attribute` is defined in line 22, and then in line 23 it is initialized using the `__ATTR`. This attribute, and thus also the file associated with it, is named `number` with access rights `0664`, i.e. read and write access for the owner of the file and the group to which it belongs, and read access for other users. The kernel will not allow regular users to modify such a file, so setting the last digit to `6` would be pointless. The `number_show()` and `number_store()` functions will handle reading and writing operations on this file.

Line 25 of the module creates a pointer to a structure of type `struct attribute`. It points to the `attr` field of `number_kattribute` structure. Its role will be explained in the description of the module constructor.

Lines 27-44 contain the module constructor code. In line 29, a kernel object is created and initialized using the `kobject_create_and_add()` function. As its first call argument, the string „test" is passed. This is the name given to the created kernel object. This will also be a name of the directory in the `sysfs` file system with which this object will be associated. As the second call argument, the address of the kernel object associated with the module is passed. Each module loaded into the kernel is represented by a `struct module` structure. One of its fields is the `mkobj` structure, which in turn contains the `kobj` kernel object as one of internal fields. This object has the same name as the file with the compiled module (without the `.ko` extension). Access to the structure representing a given module in the kernel can be obtained from its source code using the `THIS_MODULE` macro. Thus, the object associated with the object created in line 29 will be the object associated with the current module. This means that the directory named `test` will be a subdirectory of the directory named `sysfs_test` (the module will have that name) in the `sysfs` file system. Please note the exception handling. If the kernel object cannot be created, control is passed using the `goto` statement to the location in the function labelled `err`. The `printk()` function is called there, which puts an appropriate message in the kernel buffer and the constructor terminates by returning the `-ENOMEM` error code. Using `goto` instructions to handle emergencies is a common practice among kernel programmers. It is important that the label to which this instruction refers should be in the same function as the `goto` instruction associated with it. In line 33 of the constructor, the attribute indicated by `number_attribute` is associated with the created object. This is done by calling the `sysfs_create_file()` function and it means creating a file in the `sysfs` file system. The expression `&number_kattribute.attr` could be used as the second call argument to this function, but this would be less readable. Therefore, the `number_attribute` pointer is used, which contains this address, and was previously defined. If an exception occurs, the `goto` statement passes control to a place in the constructor labelled `err1`. Exception handling is similar to lines 37-39, but this time the previously created kernel object is also removed (line 41).

Lines 46-51 contain the module destructor code. In line 48 of this function, the file associated with the `number_attribute` is deleted, and in line 50 the kernel object created in the constructor is deleted.

The file associated with the `number_kattribute` is named `number` and is located in the directory under the following path: `/sys/module/sysfs_test/test`. As mentioned before the `/sys` is the directory where the `sysfs` file system is mounted. The `module` directory collects directories associated with individual modules loaded into the kernel. The `sysfs_test` directory is the directory associated with the sample module being described. The `test` directory is a directory associated with the kernel object created in this module and contains the file called `number`. The `root` user can modify this file using the `echo` shell command. Same time this file can be read by any user with the `cat` command.

Listing 10 contains the source code of another module, which uses a file in the `sysfs` file system the same way as the previous module, but it creates it in a different way. This time the module uses the object class for this.

Listing 10: Module using the sysfs with the kobj_type type

```
1 #include<linux/module.h>
2 #include<linux/sysfs.h>
3 #include<linux/kobject.h>
4 #include<linux/slab.h>
```

```c
#include<linux/string.h>

static struct kobject *kernel_object;
static int number;
static const char attribute_name[] = "number";

static ssize_t number_show(struct kobject *kernel_object, struct attribute *attribute, char *buffer)
{
        return sprintf(buffer,"%d\n",number);
}

static ssize_t number_store(struct kobject *kernel_object, struct attribute *attribute, const char *buffer, size_t count)
{
        sscanf(buffer,"%d",&number);
        return count;
}

static struct sysfs_ops operations = {
        .show = number_show,
        .store = number_store,
};

static void number_release(struct kobject *kernel_object)
{
        kfree(kernel_object);
        pr_notice("Release function activated!\n");
}

static struct attribute number_attribute = {
        .name = attribute_name,
        .mode = 0600
};

static struct attribute *attributes[] = {
        &number_attribute,
        NULL
};

static struct kobj_type ktype =
{
        .release = number_release,
        .sysfs_ops = &operations,
        .default_attrs = attributes,
};

static int __init sysfs_test_init(void)
{
        kernel_object = (struct kobject *)kmalloc(sizeof(struct kobject),GFP_KERNEL);
        if(IS_ERR(kernel_object))
                goto err1;
        memset(kernel_object,0,sizeof(struct kobject));
        kobject_init(kernel_object,&ktype);
        if(kobject_add(kernel_object,&THIS_MODULE->mkobj.kobj,"test%d",2))
                goto err2;

        return 0;
err2:
        kfree(kernel_object);
        kernel_object = NULL;
err1:
        pr_alert("Error adding a kobject\n");
        return -ENOMEM;
}

static void __exit sysfs_test_exit(void)
{
        if(kernel_object)
                kobject_put(kernel_object);
}
```

```

module_init(sysfs_test_init);
module_exit(sysfs_test_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("A second kernel module demonstrating the usage of sysfs.");
```

As in the previous module, the kernel object is created dynamically. The pointer to this object is declared in line 7. Additionally, in line 9, an array of characters is created for the file name used by the module.

The `number_show()` and `number_store()` functions are defined as in the previous module. The only difference is the type of the second call argument. This time it is a `struct attribute` instead of `struct kattribute`. Pointers to these functions are stored in the corresponding fields of the `operations` structure of type `struct sysfs_ops` (lines 22-25).

Lines 27-31 contain the definition of the `number_release()` function, which will be called when the object stops being used. This happens after calling `kobject_put()` for this object. As recommended by kernel programmers, this function should not be empty, so it releases the memory allocated to the kernel object and places the appropriate message in the kernel buffer.

In lines 33-36, a structure named `number_attribute` and type `struct attribute` is declared and initialized. The address of the array containing the attribute name is assigned to the `name` field of that structure. The access rights to this file are stored in the `mode` field. This time only the `root` user will be able to read and write to this file.

In lines 38-41, an array of attributes is created, because the initiation of one of the object's class fields requires such an array. It will contain only two elements. The first element will contain the address of the `number_attribute` structure, and the second element will be `NULL`. It means that this is the last element of this array.

Lines 43-48 contain creation and initialization of the structure called `ktype` of type `struct kobj_type`. Only three fields of this structure are initialized: the `default_attrs` field indicating the array of attributes, the `sysfs_ops` field indicating the structure containing pointers to the `show()` and `store()` methods, and the `release` field indicating the function that finalizes the object.

In the module constructor, in line 52, a kernel object is created by calling the `kmalloc()` function. If its creation fails, the `goto` statement will pass control to the line labelled `err1`, where the code that handles this exception is located. After a successful creation of kernel object, its contents are reset by calling the `memset()` function (line 55), and then it is initialized (line 56) using the `kobject_init()` function and bound to the class structure. Then, in line 57, it is given the name `test2` and it is added to the hierarchy of all kernel objects. These actions are performed by the `kobject_add()` function. If its execution fails, the `goto` statement will pass control to the constructor's place marked with the `err2` label. This code is responsible for handling this exception by releasing the memory allocated to the object and resetting the pointer to it, and then performing all the actions that they are also executed in the exception handling of the `kmalloc()` function.

In lines 69-73 a module destructor is defined. The only action performed in it is to call the `kobject_put()` function for the kernel object. Calling this function reduces the value of the reference counter for the given object. For the object used by this module the reference counter is set to `1` by calling `kobject_init()` and it is not changed afterwards. Therefore, the call to `kobject_put()` will reset this counter and thus this object will be marked as unused and removed from the kernel object hierarchy.

The file created by the module from Listing 10 has the same name as the file created by the previous module, but the path to it is slightly different: `/sys/module/sysfs_test2/test2`. The use of this file has remained unchanged, except that only the `root` user has read and write access on it.

Examples of other modules that use the `sysfs` file system can be found in the Linux kernel sources in the `samples/kobject` subdirectory. The `kset-example.c` file contains the module source code that uses the user event mechanism.

Exercises

1. `[3 points]` Change the module source code from Listing 3 so that only one value is read from the file. Use the `single_open()` and `single_release()` functions. Note that in this case you only need to implement the `show()` function.

2. `[5 points]` Change the module source code from Listing 3 so that it does not use the sequential file mechanism to perform the reading.

3. `[7 points]` Change the module source code from Listing 3 so that the buffer for the file is a bidirectional list.

4. `[3 points]` Remove the `goto` instruction from the module constructor from Listing 10, so that the behavior of this function remains unchanged in case of an exception.

5. `[5 points]` Change the module source code from Listing 9 or 10 so that it creates a directory with a file directly in the `/sys` directory.

6. `[7 points]` Write a module that will create a file in the module's subdirectory (under the `/sys` path) into which the user will be able to write only natural numbers from `1` to `7`. All other values should be ignored. Once the correct number is entered, the module should create as many new files in the subdirectory as the given value is. Names of those new files can be created by adding continuous numbers to the primary file name. The module should store random numbers in the files. Files along with the subdirectory should be removed by the module destructor.