Laboratory 1: "Linux kernel modules" (one week)

dr inż. Arkadiusz Chrobot
dr inż. Karol Tomaszewski

March 3, 2024

Contents

Introduction 1

1. A simple Linux kernel module 1

2. Compilation of the module 3

3. Handling the modules 4

4. Parameters of the kernel modules 5

5. The `printk()` function 8

6. Data types 8

7. Helpful macros 9

Exercises 10

Introduction

This guide provides information on how to create and compile modules for Linux kernel version 4.15[1]. It also describes the specifics of software development for the kernel space. Chapter 1 contains the source code for a simple kernel module, along with its description. Chapter 2 describes how to compile the kernel module. Next, Chapter 3 is dedicated to commands that are used to handle modules (loading, removing, etc.). Chapter 4 describes how to use kernel module parameters. Chapter 5 provides more details about the `printk()` function. The following Chapter 6 describes data types specific to Linux kernel modules. Finally, Chapter 7 provides information on useful macros that have been defined by Linux programmers. The document ends with a list of tasks to be done during the laboratory.

1. A simple Linux kernel module

Kernel modules are files that contain executable code that can be dynamically included in the kernel. In other words, they are the equivalent of dynamically linked libraries used by programs in user space. Modules are used to implement mechanisms that extend the functionality of the kernel. Most often these mechanisms are device drivers, but they can also be network packet filters or file system implementations. A source code for the simple module that generates the *Hello World* message when loading and removing from the kernel is presented in Listing 1.

Listing 1: Simple Linux kernel module

```
1  #include<linux/module.h>
2
3  static int __init first_init(void)
4  {
5          printk(KERN_ALERT"Welcome\n");
6          return 0;
7  }
8
9  static void __exit first_exit(void)
10 {
11         printk(KERN_ALERT"Good bye\n");
```

[1]Laboratory instructions were created based on this version of the Linux kernel. The information contained therein, particularly the source code of kernel modules, is up-to-date with this version, but not necessarily with the newer ones.

```
12 }
13
14 module_init(first_init);
15 module_exit(first_exit);
16 MODULE_LICENSE("GPL");
17 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
18 MODULE_DESCRIPTION("Another \"Hello World!\" kernel module :-)");
19 MODULE_VERSION("1.0");
```

Software libraries that are used in user space cannot be used at the kernel level of the operating system. The kernel has its own libraries, and therefore also has its own set of header files. The `module.h` module, included in the first line of the module code, contains tags, macros and other elements that are necessary to create even the simplest kernel module. Two functions are defined in the module source code. Both are static functions (please note the `static` keyword in their headers), which means that their names are visible only inside the module. Most functions in the module should be defined in such a way that their function names do not collide with the names of other kernel functions. The exceptions are the functions that we want to make available outside of the module so that other modules can use them. This topic will be described in Chapter 7. The functions responsible for loading and removing the module should be pointed to the compiler using the `module_init` and `module_exit` macros, respectively (lines 14 and 15). Both take function names as call arguments, which can be any of the names allowed by C language rules. The initialization function must return an `int` value, which means its exit code. The correct ending of this function is indicated by zero and incorrect ending is indicated by a negative number. It takes no call arguments. In its prototype, the `__init` tag can be used, which informs the kernel that this function will be used only at the beginning of the module and only once. Thanks to this, kernel can free memory for this function just after it is executed. The removal function is called just before removing the module from the kernel and its task is to „clean up" after the module. This may mean performing various actions, depending on what role the module played in the kernel, e.g. if it is a device driver, then its removal function may turn off the device. Like the initial function, the removal function has no parameters, but unlike the first one, it also doesn't return any value. It may be tagged with `__exit` (line 9), which informs the kernel that this function will only be used if the module is removed from the kernel. This is important when the module is permanently included in the kernel, i.e. at the compilation stage, or when the kernel is compiled with an option that disables module removal. In such cases, the function may be omitted when the module is included in the kernel. The main task of both functions in the module from Listing 1 is to put a message in the `kernel buffer` using the `printk()` function. The `printk()` function is similar in use to the `printf()` function, however, to check its output, the `dmesg` command can be used or the content of appropriate file in `/var/log` directory can be viewed. Depending on Linux distribution, this file will be called `kern.log` or `messages`. In both cases it is a text file. Some Linux distributions additionally print the kernel buffer content to the console by default (monitor screen in most cases). In other distributions, this behavior can be enabled using the `dmesg` command mentioned above. It is also worth noting that the string passed to the `printk()` function call is preceded by the `KERN_ALERT` constant. It determines what priority the message placed in the kernel buffer will have. A description of the `printk()` function and related commands will be provided later in the guide number 5. Lines 16-19 of the module code contain macro calls that specify the license for which the module is available, and also contain the basic description of the module. If the `MODULE_LICENSE` macro is not used, or is invoked with a string denoting a license not belonging to the free license, then the kernel after loading the module will generate a message that it has been „contaminated" with a proprietary module. Such information will be included in every *kernel oops* message to inform those investigating the causes of a kernel failure that it may have its cause in code that is accessible only in binary form[2]. Other macros can be omitted without major consequences, however, their use makes it easier for system administrators to determine what a module is useful for and how to use it.

[2]This usually means that Linux kernel developers will not attempt to determine the cause of the failure.

2. Compilation of the module

The kernel has its own compilation system, called *kbuild*, which is also used to build modules. More information about it can be found in the kernel documentation located in the `/usr/src/linux/Documentation/kbuild/` directory. The kernel module compilation requires the installation of the source files for the kernel version used on the system and the use of the `make` tool. The execution of the `make` command can be determined by creating the appropriate configuration file. Listing 2 shows the content of such a file. Its content seems to be complicated, but it can be considered as a template that can be used many times, changing only its first line.

Listing 2: Makefile for compiling the sample kernel module

```
obj-m := first.o
KERNELDIR = /lib/modules/$(shell uname -r)/build

default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

The configuration file shown in Listing 2 assumes that the module code from Listing 1 is saved in the file `first.c`. To compile it, just put its name with the extension `.o` in the first line of the `Makefile`. The `obj-m` notation means that the resulting file will be a module, and will not be included in the compiled form of the kernel. The `KERNELDIR` variable contains the path to the directory where the kernel headers are stored. The `$(shell uname -r)` notation means that the `uname` shell command will be invoked with the `-r` option to determine the kernel version. The string corresponding to this version is also the name of the directory that contains the existing kernel modules, along with the link to the (`build`) directory. After the `make` command is called, the `default` rule is executed. The variable `MAKE` contains the name of the `make` program, the variable `PWD` contains the current directory in which the source code of the compiled module should be located. The rule `clean` allows to delete files created during module compilation, including the file containing the compiled module. To run it, the `make` command should be called as follows:

```
make clean
```

To compile a module with a different name, it is enough to change the name in the first line. To compile a module consisting of several source files, additional lines have to be added. For example, if the module code is contained in files named `first.c` and `second.c`, and you want to call it `third`, then the following entry should be placed at the beginning of the `Makefile`:

```
obj-m := third.o

third-objs := first.o second.o
```

Listings 3 and 4 contain an example of the source code of the module divided into two files, and Listing 5 contains the content of the `Makefile`, which allows to compile it and save the result of this compilation in the file `hello.ko`.

Listing 3: File containing the first part of the source code of the sample kernel module

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

static int __init first_init(void)
{
	printk(KERN_ALERT"Welcome!");
	return 0;
```

```
}

module_init(first_init);
```

Listing 4: File containing the second part of the source code of the sample kernel module

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

static void __exit first_exit(void)
{
        printk(KERN_ALERT"Good bye!");
}

module_exit(first_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_DESCRIPTION("\"Hello world\" module in two parts.");
MODULE_VERSION("1.0");
```

Listing 5: Makefile file with instructions compiling the source code of the module split into two files

```
obj-m := hello.o
hello-objs := first.o second.o

KERNELDIR = /lib/modules/$(shell uname -r)/build

default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

To compile many modules with one configuration file, the first line of it should be replaced with a configuration similar to the one below:

```
obj-m += mod1.o

obj-m += mod2.o

obj-m += mod3.o
```

In the above example, three modules will be compiled, whose source is contained in the files named `mod1.c`, `mod2.c` and `mod3.c`. If no errors are found in the source code of the module during compilation, then it will result with a file with the extension .ko, which will contain the resulting code of the module.

3. Handling the modules

The compiled module can be included (loaded) into the kernel by the system administrator (root user) or a user with appropriate permissions. The system shell provides several commands that allow to do this work and other tasks related to managing kernel modules. The simplest command to load a module to the kernel is `insmod`. The only required call argument to invoke this command is the name, along with the extension, of the file containing the compiled module. If the command is issued by a user with appropriate permissions, like the `so2` user, then it has to be prefixed by the `sudo` command. For example, for the module described in Chapter 1, the `insmod` call can have the following form:

```
sudo insmod first.ko
```

Optionally, if the module accepts its own call arguments, their values can be listed after the module name in the `insmod` command. Another command to load the module into the kernel is `modprobe`. Compared to `insmod`, the `modprobe` command also loads all other modules associated with the one listed on the command line. Let's assume that the `first.ko` module is in the current directory. The syntax for loading this module with the `modprobe` command is as follows:

```
sudo modprobe ./first.ko
```

The `modprobe` command can also be used to perform other operations related to module support. Details of this command can be found in the system manual (`man modprobe`).
The list of modules loaded into the kernel is displayed by the `lsmod` command. In addition, the `lsmod` command provides the information about the amount of memory used by each module and how many other modules use the functionality provided by every module. This command does not take any call arguments. It works by formatting and displaying the information contained in the file (`/proc/modules`).
The `modinfo` command displays the information about the module saved in the file with the „.ko" extension. To get information about the `first.ko` module, the `modinfo` command should be called as follows:

```
modinfo first.ko
```

By default, all the information that the author placed in the module using the appropriate macros are displayed. Additional options for this command are described in the system manual: `man modinfo`.
The module can be removed from memory by the `rmmod` command. To remove a module, use the `rmmod` command with the module name:

```
sudo rmmod first
```

Additional options for this command are described in the system manual: `man rmmod`.

4. Parameters of the kernel modules

Module parameters are variables that its values can be set when loading module into memory. In the module whose code is placed in Listing 6 there are three such variables of type `int`. They are first declared as static variables that have a specific initial value. Then each of them is changed into a parameter using the `module_param` macro, which takes three call arguments: variable name, variable type name and number specifying the access rights to this variable. In the case of the described module, it is the read and write right for the owner and read right for other users. The parameter value can be modified in the module code, as it happens in the destructor of an example module. The `MODULE_PARM_DESC` macro definition is used to describe module parameters. It takes two arguments - the name of the parameter and a string representing its description, which can be seen after compiling the module and using the `modinfo` command on the resulting file.

Listing 6: Simple kernel module with parameters

```
1  #include<linux/module.h>
2
3  static int foo = 1;
4  static int bar = 2;
5  static int baz = 3;
6
7  module_param(foo,int,0644);
8  MODULE_PARM_DESC(foo,"parameter");
9  module_param(bar,int,0644);
10 MODULE_PARM_DESC(bar,"parameter");
```

```
11 module_param(baz,int,0644);
12 MODULE_PARM_DESC(baz,"parameter");
13
14 static int __init parammod_init(void)
15 {
16         printk(KERN_ALERT "Module parameters:\n");
17
18         printk(KERN_ALERT "foo value: %d\n",foo);
19         printk(KERN_ALERT "bar value: %d\n",bar);
20         printk(KERN_ALERT "baz value: %d\n",baz);
21
22         return 0;
23 }
24
25 static void __exit parammod_exit(void)
26 {
27         printk(KERN_ALERT "Module parameters:\n");
28
29         foo++; bar++; baz++;
30
31         printk(KERN_ALERT "foo value: %d\n",foo);
32         printk(KERN_ALERT "bar value: %d\n",bar);
33         printk(KERN_ALERT "baz value: %d\n",baz);
34
35 }
36
37 module_init(parammod_init);
38 module_exit(parammod_exit);
39
40 MODULE_LICENSE("GPL");
41 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
42 MODULE_DESCRIPTION("Kernel module with parameters.");
43 MODULE_VERSION("1.0");
```

Module from Listing 6 can be loaded into kernel in a similar way as previously described:

```
sudo insmod parmmod.ko
```

Then the parameters `foo`, `bar` and `baz` will have the value that was given to them at the place of declaration. When removing the module from memory, the value will only be increased by `1`. The module can also be loaded giving a different parameter values. For example:

```
sudo insmod parmmod.ko foo=3 bar=4 baz=5
```

will change the initial values of all three parameters, and the command:

```
insmod parmmod.ko foo=5
```

will give an initial value of `5` only to the `foo` parameter.

The parameter can also be a character string or an array. To create a parameter through which a string can be passed, an array of `char` elements have to be declared at first, and then the `module_param_string` macro have to be used, which takes four arguments. The first argument is the name of the parameter, the second argument is the name of the array, the third argument is the maximum number of characters that the array can contain, and the fourth argument is the octal number that defines the access rights to the parameter. For an array parameter, the `module_param_array` macro have to be used instead of the

`module_param` macro. It also accepts four arguments: the name of the array, the type of array elements, the address of the `int` type variable, in which the macro will store the number of array elements provided while loading, and the last argument is the octal number that defines the parameter access rights. Listing 7 contains the module code to which a character string and array values can be passed through parameters while loading.

Listing 7: Kernel module with character string and array parameters

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

#define ARRAY_NUMBER_OF_ELEMENTS 10
#define STRING_NUMBER_OF_ELEMENTS 40

static char foo[STRING_NUMBER_OF_ELEMENTS];
static int array[ARRAY_NUMBER_OF_ELEMENTS];
static int number_of_elements = 0;

module_param_string(foostring,foo,STRING_NUMBER_OF_ELEMENTS,0644);
MODULE_PARM_DESC(foo,"A char array parameter");
module_param_array(array,int,&number_of_elements,0644);
MODULE_PARM_DESC(array, "An array parameter");

static int __init parammod_init(void)
{
        int i;
        printk(KERN_ALERT"Module parameter:\n");

        printk(KERN_ALERT"foo value: %s\n",foo);

        printk(KERN_ALERT"array values: ");
        for(i=0;i<number_of_elements;i++)
                printk(KERN_CONT"%d\n",array[i]);

        return 0;
}

static void __exit parammod_exit(void)
{
        int i;
        printk(KERN_ALERT "Module parameter:\n");

        printk(KERN_ALERT "foo value: %s\n",foo);

        printk(KERN_ALERT"array values: ");
        for(i=0;i<number_of_elements;i++)
                printk(KERN_CONT"%d\n",array[i]);
}

module_init(parammod_init);
module_exit(parammod_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_DESCRIPTION("Kernel module with char array (string) and array parameters.");
MODULE_VERSION("1.0");
```

To give values to the parameters `foostring` and `array`, the compiled module can be loaded as follows:

```
sudo insmod ./parmcharmod.ko foostring="Science" array=1,2,3,4,5,6
```

5. The `printk()` function

The `printk()` function, as mentioned in Chapter 1, works similar to the `printf()` function. Yet, it does not print information directly to the screen, but puts it in the kernel buffer, from where, depending on the system configuration, they go to the appropriate files. The contents of this buffer can be displayed on the screen using the `dmesg` command. A useful option that comes with the `dmesg` command is `-c`, which causes the command to print the contents of the buffer and clean the buffer afterwards. When using this option the newly incoming kernel messages will go to an empty buffer and will be easier to find. More `dmesg` options can be found under: `man dmesg`. As explained in Chapter 1, the character string containing the massage in `printk()` function call is preceded by a constant variable that specifies the logging level, i.e. the priority of this information. **No other character is allowed between this constant variable and the message string except the white space character, and even this character is not necessary.** To make the `printk()` function easier to use, kernel programmers have defined a number of functions with short names that call `printk()` function with the appropriate log level. Table 1 describes the fixed log levels and lists the names of the functions mentioned.

Table 1: Log level variables and `printk()` function aliases (based on `http://elinux.org/Debugging_by_printing`)

Name	Description	Alias
`KERN_EMERG`	Highest log level for exceptions that may cause system crashes or instability.	`pr_emerg()`
`KERN_ALERT`	Log level for events that require immediate attention of the user.	`pr_alert()`
`KERN_CRIT`	Log level for critical events related to software or hardware failure.	`pr_crit()`
`KERN_ERR`	The log level for errors, most often used by device drivers to report hardware problems.	`pr_err()`
`KERN_NOTICE`	Log level for events, e.g. related to security, which usually do not have serious consequences, but are worth noting.	`pr_notice()`
`KERN_INFO`	Log level for regular information.	`pr_info()`
`KERN_DEBUG`	Log level for debug related information.	`pr_debug()`
`KERN_DEFAULT`	Default log level.	
`KERN_CONT`	If the string in the previous `printk()` call did not end in a new line character (`\n`), then use this log level in the next call.	`pr_cont()`

Most of the sample kernel modules presented in this guide use the `KERN_ALERT` log level so that information from these modules is put into the kernel buffer as soon as possible.

6. Data types

When creating kernel modules, all the basic types defined in C language can be used.

However, since the C language standard does not precisely specify the size of these data types, and the kernel and hence its modules need to work the same on different hardware platforms, Linux developers have developed special basic data types whose size is independent of the hardware platform and is always same. They are available after including the `linux/types.h` header in the kernel module code. Their description is given in Table 2. Although floating point types such as `float` and `double` can be used in the kernel source code, no operators are defined for them. Their operation involves the use of a floating point unit (FPU), which is quite computational expensive from the kernel's point of view. Additionally, floating point arithmetic is not needed in most cases in the kernel space. In case it is necessary to use

Table 2: Basic data types for the kernel

Name	Description
s8	Integer number, 8-bit.
u8	Natural number, 8-bit.
s16	Integer number, 16-bit.
u16	Natural number, 16-bit.
s32	Integer number, 32-bit.
u32	Natural number, 32-bit.
s64	Integer number, 64-bit.
u64	Natural number, 64-bit.

it, then expressions containing variables and floating point operators should be placed between calls to the `kernel_fpu_begin()` and `kernel_fpu_end()`. Also the `-mhard-float` compilation option should be used. The Linux kernel also has substitutes for other functions for handling the character strings that are known from user space. Usually they have the same name and accept the same arguments. The difference is that to use them the `linux/string.h` header file should be included in the module code, instead of `string.h`. The `sprintf()` function is also declared in the same header file as the `printk()` function, and it works just like its equivalent from user space.

7. Helpful macros

If the message printed by `printk()` relates to a runtime exception, the location of its cause may be facilitated by the use of the `__LINE__` and `__FILE__` macros. Kernel modules API changes quite often from version to version. The `LINUX_VERSION_CODE` and `KERNEL_VERSION` macros come in handy for writing a version-independent code of kernel module. The first one returns a number that represents the kernel version number for which the module was compiled. It takes no call arguments. The second macro takes three call arguments specifying the version number and converts them into a single number, such as the format returned by `LINUX_VERSION_CODE` (the kernel version number in hexadecimal code). Thanks to this, these two numbers can be compared with each other. Both macros are defined in the `linux/version.h` header file. Listing 8 presents one special use case of those macros.

Listing 8: Example of using macros for version management

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

static int __init kernelinfo_init(void)
{
        if(LINUX_VERSION_CODE>KERNEL_VERSION(2,6,0))
                printk(KERN_ALERT "This is not kernel 2.6.0.\
                        It's number in hexdecimal is: %x",LINUX_VERSION_CODE);
        return 0;
}

static void __exit kernelinfo_exit(void)
{
        if(LINUX_VERSION_CODE>KERNEL_VERSION(2,6,0))
                printk(KERN_ALERT "This is not kernel 2.6.0.\
                        It's number in hexdecimal is: %x",LINUX_VERSION_CODE);
}

module_init(kernelinfo_init);
module_exit(kernelinfo_exit);
```

```
21
22 MODULE_LICENSE("GPL");
23 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
24 MODULE_DESCRIPTION("Kernel module that informs about kernel version");
25 MODULE_VERSION("1.0");
```

In the presented module, the constructor and destructor use the macros described earlier to determine whether they were run with a kernel version higher than *2.6.0*. If so, they call `printk()` function, which puts an appropriate message in the kernel buffer, containing the value of the real kernel version number in hexadecimal code. It is worth to note that in this example the `LINUX_VERSION_CODE` and `KERNEL_VERSION` macros are usually used together with preprocessor instructions such as `#ifndef` or `#ifdef`, while in the given example they are used directly in the kernel module code. If we create header files in which we mix code for kernel with code for user mode, the `__KERNEL__` macro may be useful to separate them. It is used, for example, in the `#ifdef` statement to indicate to the compiler which code to treat as kernel code. The `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL` macros are used to share variables and functions defined in the header file or file with the source code of the module to other modules. In case of sharing the variables defined in the header file the variable declaration should be preceded by the `extern` keyword. Those macros take only one call argument, which is the name of the variable or function. The difference in their operation is that the names provided via `EXPORT_SYMBOL_GPL` are only available for modules under the GPL license.

Exercises

1. `[3 points]` Use the `sizeof` operator to examine the sizes of the data types described in Table 2.
2. `[5 points]` Demonstrate the use of functions `strscpy()`, `strlcat()` and `strncmp()`, that operate on strings, in the kernel module. Use the Linux Cross-Reference website, to find comments in the kernel code that document these functions.
3. `[7 points]` Write modules that will demonstrate the use of the `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL` macros.
4. `[3 points]` Write a kernel module in which you assign the variable of the `double` type with a value of `0.1*0.1`. Check if this module will compile. Find an explanation for the result of this experiment.
5. `[5 points]` Write a kernel module in which you will demonstrate the `__FILE__` and `__LINE__` macros.
6. `[7 points]` Write a function which - depending on whether it will be included in a regular program or in the kernel module - will write an appropriate message about which mode it was called in. Use the `__KERNEL__` macro.