A Simple Driver for Linux OS

apriorit **Apriorit Inc, Danil Ishkov**, 27 Sep 2010

4.90 (44 votes)

In this article, I am going to describe the process of writing and building of a simple driver-module for Linux OS

Download driver - 2.19 KB

Table of Contents

- Introduction
- General Information
- Functions of Module Loading and Unloading
- Registration of the Character Device
- The Usage of Memory Allocated in the User Mode
- The Kernel Module Build System
- The Module Loading and Its Usage
- Bibliography List

Introduction

In this article, I am going to describe the process of writing and building of a simple driver-module for Linux OS. Meanwhile, I will touch upon the following questions:

- The system of the kernel logging
- The work with character devices
- The work with the “user level” memory from the kernel

The article concerns the Linux kernel version 2.6.32 because other kernel versions can have the modified API, which is used in examples or in the build system.

General Information

Linux is a monolithic kernel. That is why the driver for it should be compiled together with the kernel itself or should be implemented in the form of a kernel module to avoid the recompiling of the kernel when driver adding is needed. This article deals with the kernel modules exactly.

A module is an object file prepared in a special way. The Linux kernel can load a module to its address space and link the module with itself. The Linux kernel is written in 2 languages: C and assembler (the architecture dependent parts). The development of drivers for Linux OS is possible only in C and assembler languages, but not in C++ language (as for the Microsoft Windows kernel). It is connected with the fact that the kernel source pieces of code, namely, header files, can contain C++ key words such as `new`, `delete` and the assembler pieces of code can contain the `‘::’` lexeme.

The module code is executed in the kernel context. It rests some additional responsibility in the developer: if there is an error in the user level program, the results of this error will affect mainly the user program; if an error occurs in the kernel module, it may affect the whole system. But one of the specifics of the Linux kernel is a rather high resistance to errors in the modules’ code. If there is a non-critical error in a module (such as the dereferencing of the null pointer), the `oops` message will be displayed (`oops` is a deviation from the normal work of Linux and in this case, the kernel creates a log record with the error description). Then, the module, in which the error appeared, is unloaded, while the kernel itself and the rest of modules continue working. However, after the `oops` message, the system kernel can often be in an inconsistent state and the further work may lead to the kernel panic.

The kernel and its modules are built into a practically single program module. That is why it is worth remembering that within one program module, one global name space is used. To clutter up the global name space minimally, one should monitor that the module exports only the necessary minimum of global characters and that all exported global characters have the unique names (the good practice is to add the name of the module, which exports the character, to the name of the character as a prefix).

Functions of Module Loading and Unloading

The piece of code that is required for the creation of the simplest module is very simple and laconic. It looks as follows:

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

static int my_init(void)
{
                       return  0;
}

static void my_exit(void)
{
                       return;
}

module_init(my_init);
module_exit(my_exit);
```

This piece of code does not do anything but allowing loading and unloading the module. When loading the driver, the `my_init` function is called; when unloading the driver, the `my_exit` function is called. We inform the kernel about it with the help of the `module_init` and `module_exit` macros. These functions must have exactly the following signature:

```
int init(void);
void exit(void);
```

The linking of the ***linux/module.h*** header file is necessary for adding information about a kernel version, for which the module is built, to the module itself. Linux OS will not allow loading of the module that was built for another kernel version. It is because the kernel API changes intensively and the change of signature of one of the functions used in the module will lead to the damage of the stack when calling this function. The ***linux/init.h*** header file contains the declaration of the `module_init` and `module_exit` macros.

Registration of the Character Device

We will not dwell on such a simple module. I would like to demonstrate the work with the device files and with logging in the kernel. These are tools that will be useful for each driver and will somewhat expand the development in the kernel mode for Linux OS.

First, I would like to say a few words about the device file. The device file is a file that is usually located in hierarchy of the ***/dev/*** folder. It is the easiest and the most accessible way of interaction of the user code and the kernel code. To make it shorter, I can say that everything that is written to such file is passed to the kernel, to the module that serves this file; everything that is read from such file comes from the module that serves the file. There are two types of device files: character (non-buffered) and block (buffered) files. The character file implies the possibility to read and write information to it by one character whereas the block file allows reading and writing only the data block as a whole. This article will touch upon only the character device files.

In Linux OS, device files are identified by two positive numbers: **major device number** and **minor device number**. The **major device number** usually identifies the module that serves the device file or a group of devices served by a module. The **minor device number** identifies a definite device in the range of the defined **major device number**. These two numbers can be either defined as constants in the driver code or received dynamically. In the first case, the system will try to use the defined numbers and if they are already used, it will return an error. Functions that allocate the device numbers dynamically also reserve the allocated device numbers so that the dynamically allocated device number cannot be used by another module when it is allocated or used.

To register the character device, the following function can be used:

```
int register_chrdev (unsigned int   major,
                     const char *   name,
                     const struct   fops);
                     file_operations *
```

It registers the device with the specified name and major device number (or it allocates the major device number if the `major` parameter is equal to zero) and links the `file_operations` structure with the device. If the function allocates the major device number, the returned value will be equal to the allocated number. In other case, the zero value means the successful completion and the negative value means an error. The registered device is associated with the defined major device number and minor device number is in the range of 0 to 255.

The string that is passed as the `name` parameter is the name of the device or the module if the last registers only one device and is used for the identification of the device in the ***/sys/devices*** file. The `file_operations` structure contains the pointers to the functions that must process the manipulations with the device file (such as open, read, write, etc.) and the pointer to the `module` structure that identifies the module, which implements these functions. The structure for the kernel version 2.6.32 looks as follows:

```
struct file_operations {
       struct module *owner;
       loff_t (*llseek) (struct file *, loff_t, int);
       ssize_t (*read) (struct file *, char *, size_t, loff_t *);
       ssize_t (*write) (struct file *, const char *, size_t,
loff_t *);
       int (*readdir) (struct file *, void *, filldir_t);
       unsigned int (*poll) (struct file *, struct
poll_table_struct *);
       int (*ioctl) (struct inode *, struct file *, unsigned
int, unsigned long);
       int (*mmap) (struct file *, struct vm_area_struct *);
       int (*open) (struct inode *, struct file *);
       int (*flush) (struct file *);
       int (*release) (struct inode *, struct file *);
       int (*fsync) (struct file *, struct dentry *, int
datasync);
       int (*fasync) (int, struct file *, int);
       int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *,
unsigned long,
          loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *,
unsigned long,
          loff_t *);
    };
```

It is not necessary to implement all functions from the `file_operations` structure to use the file. If the function is not implemented, the corresponding pointer can be of zero value. In this case, the system will implement some default behavior for this function. It is enough to implement the `read` function for our example.

As our driver will provide the work of devices of one type, we can create the global `static file_operations` structure and fill it statically. It can look as follows:

```
static struct file_operations simple_driver_fops =
{
    .owner   = THIS_MODULE,
    .read    = device_file_read,
};
```

Here, the `THIS_MODULE` macro (declared in ***linux/module.h***) will be converted to the pointer to the `module` structure that corresponds to our module. The `device_file_read` is a pointer to the function with the prototype, whose body we will write later.

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

So, when we have the `file_operations` structure, we can write a pair of functions for registration and unregistration of the device file:

```
static int device_file_major_number = 0;
static const char device_name[] = "Simple-driver";

static int register_device(void)
{
        int result = 0;

        printk( KERN_NOTICE "Simple-driver: register_device() is
called." );

        result = register_chrdev( 0, device_name,
&simple_driver_fops );
        if( result < 0 )
        {
            printk( KERN_WARNING "Simple-driver:  can\'t
register
                character device with errorcode = %i", result );
            return result;
        }

        device_file_major_number = result;
        printk( KERN_NOTICE "Simple-driver: registered character
device
            with major number = %i and minor numbers 0...255"
             , device_file_major_number );

        return 0;
}
```

We store the major device number in the `device_file_major_number` global variable as we will need it for the device file unregistration in the end of the "driver life".

In the listing above, the only function, which was not mentioned, is the `printk()` function. It is used for logging of messages from the kernel. The `printk()` function is declared in the *linux/kernel.h* file and works like the `printf` library function except one nuance. As you have already noticed, each format string of `printk` in this listing has the `KERN_SOMETHING` prefix. It is the message priority and it can be of eight levels, from the highest zero level (`KERN_EMERG`), which informs that the kernel is unstable, to the lowest seventh level (`KERN_DEBUG`).

The `string` that is formed by `printk` function is written to the circular buffer. From there, it is read by the `klogd` daemon and gets to the system log. The `printk` function is written in such a way that it can be called from any place in the kernel. The worst that can happen is circular buffer overflow when the oldest messages will not get to the system log.

Now, we need only to write the function for the device file unregistration. Its logic is simple: if we succeed in the device file registration, the `device_file_major_number` value will not be zero and we will be able to unregister it with the help of the `unregister_chrdev` function declared in ***linux/fs.h***. The first parameter is the **major device number** and the second is the device name string. The `unregister_chrdev` function, by its action, is fully symmetric to the `register_chrdev` function.

We receive the following piece of code for the device registration:

```
void unregister_device(void)
{
    printk( KERN_NOTICE "Simple-driver: unregister_device() is
called" );
    if(device_file_major_number != 0)
    {
        unregister_chrdev(device_file_major_number, device_name);
    }
}
```

The Usage of Memory Allocated in the User Mode

We need to write the function for reading characters from the device. It must have the signature that is appropriate for the signature from the `file_operations` structure:

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

The first parameter of this function is the pointer to the `file` structure from which we can find out the details: what file we work with, what private data is associated with it, etc. The second parameter is a buffer that is allocated in the user space for the read data. The third parameter is the number of bytes to be read. The fourth parameter is the offset (position) in the file, starting from which we should count bytes. After the performing of the function, the position in the file should be refreshed. Also the function should return the number of successfully read bytes.

One of the actions that our read function should perform is the copying of the information to the buffer allocated by the user in the address space of the user mode. We cannot just dereference the pointer from the address space of the user mode because the address, to which it refers, can have another value in the kernel address space. There is a special set of functions and macros (declared in ***asm/uaccess.h***) for working with pointers from the user address space. The `copy_to_user()` function is the best for our task. As it can be seen from its name, it copies data from the buffer in the kernel to the buffer allocated by the user. Besides, the `copy_to_user()` function checks the pointer validity and the sufficiency of the size of the buffer allocated in the user space. It makes it easier to process errors in the driver. The `copy_to_user` prototype looks like the following:

```
long copy_to_user( void __user  *to, const void * from, unsigned
long n );
```

The first parameter, which should be passed to the function, is the user pointer to the buffer. The second parameter should be the pointer to the data source, the third – the number of bytes to be copied. The function will return 0 in case of success and not 0 in case of error. The `__user` macro in the function prototype is used for documenting. It also allows analyzing the piece of code for the correctness of using the pointers from the user address space by means of the `sparse` static code analyzer. The pointers from the user address space should always be marked as `__user`.

We create only an example of the driver and we do not have the real device. So it will be sufficient if reading from our device file will always return some text string (e.g., Hello world from kernel mode!).

Now, we can start writing the piece of code of the `read` function:

```
static const char    g_s_Hello_World_string[] = "Hello world
from kernel mode!\n\0";
static const ssize_t g_s_Hello_World_size =
sizeof(g_s_Hello_World_string);

static ssize_t device_file_read(
                        struct file *file_ptr
                       , char __user *user_buffer
                       , size_t count
                       , loff_t *position)
{
    printk( KERN_NOTICE "Simple-driver:
        Device file is read at offset = %i, read bytes count =
%u"
                , (int)*position
                , (unsigned int)count );

    /* If position is behind the end of a file we have nothing
to read */
```

```
    if( *position >= g_s_Hello_World_size )
        return 0;

    /* If a user tries to read more than we have, read only
    as many bytes as we have */
    if( *position + count > g_s_Hello_World_size )
        count = g_s_Hello_World_size - *position;

    if( copy_to_user(user_buffer, g_s_Hello_World_string +
*position, count) != 0 )
        return -EFAULT;

    /* Move reading position */
    *position += count;
    return count;
}
```

The Kernel Module Build System

Now, when the whole driver piece of code is written, we would like to build it and see how it will work. In the kernels of version 2.4, to build the module, the developer had to prepare the compilation environment himself and to compile the driver with the help of the **GCC** compiler. As a result of the compilation, the received ***.o*** file is the module loadable to the kernel. Since then, the order of the kernel modules build has changed. Now, the developer should only write a special **makefile** that will start the kernel build system and will inform the kernel what the module should be built of. To build a module from one source file, it is enough to write the one-string **makefile** and to start the kernel build system:

```
obj-m := source_file_name.o
```

The module name will correspond to the source file name and the module itself will have the ***.ko*** extension.

To build the module from several source files, we should add one `string`:

```
obj-m := module_name.o
module_name-objs := source_1.o  source_2.o … source_n.o
```

We can start the kernel build system with the help of the `make` command:

```
make –C KERNEL_MODULE_BUILD_SYSTEM_FOLDER  M=`pwd` modules
```

for the module build and

```
make –C KERNEL_MODULES_BUILD_SYSTEM_FOLDER  M=`pwd` clean
```

for the build folder cleanup.

The module build system is usually located in the ***/lib/modules/`uname -r`/build*** folder. We should prepare the module build system for building to build the first module. To do this, we should go to the build system folder and execute the following:

```
#> make modules_prepare
```

Let’s unite this knowledge into a single makefile:

```
TARGET_MODULE:=simple-module

# If we are running by kernel building system
ifneq ($(KERNELRELEASE),)
    $(TARGET_MODULE)-objs := main.o device_file.o
    obj-m := $(TARGET_MODULE).o

# If we running without kernel build system
else
    BUILDSYSTEM_DIR:=/lib/modules/$(shell uname -r)/build
    PWD:=$(shell pwd)


all :
# run kernel build system to make module
    $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) modules

clean:
# run kernel build system to cleanup in current directory
    $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) clean

load:
    insmod ./$(TARGET_MODULE).ko

unload:
    rmmod ./$(TARGET_MODULE).ko

endif
```

The `load` and `unload` targets are for loading of the built module and for deleting it from the kernel.

In our example, the driver is compiled from two files with the ***main.c*** and ***device_file.c*** source pieces of code and has the ***simple-module.ko*** name.

Module Loading and Its Usage

When our module is built, we can load it by executing the following command in the folder with the source files:

```
#> make load
```

After that, a `string` with the name of our driver appears in the special ***/proc/modules*** file. And a `string` with the device, registered by our module, appears in the special ***/proc/devices*** file. It will look as follows:

```
Character devices:
1 mem
4 tty
4 ttyS
…
250 Simple-driver
…
```

The number before the device name is a **major number** associated with it. We know the range of **minor numbers** for our device (0...255) and that is why we can create the device file in the ***/dev*** virtual file system:

```
#> mknod /dev/simple-driver c  250 0
```

When the device file is created, we will check if everything works correctly and will display its contents with the help of the `cat` command:

```
$> cat /dev/simple-driver
Hello world from kernel mode!
```

Bibliography List

- Jonathan Corbet, Alessandro Rubini,Greg Kroah-Hartman Linux Device Drivers, Third Edition, O'Reilly, ISBN 978-0-596-00590-0 http://lwn.net/Kernel/LDD3/
- Peter Jay Salzman Ori Pomerantz The Linux Kernel Module Programming Guide http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html
- Linux Cross Reference http://lxr.free-electrons.com/ident

License

Share

About the Authors

Apriorit Inc

Apriorit Inc.
Hungary

AprioriT is a Software Research and Development company that works in advanced knowledge-intensive scopes.

Company offers integrated research&development services for the software projects in such directions as Corporate Security, Remote Control, Mobile Development, Embedded Systems, Virtualization, Drivers and others.

Official site http://www.apriorit.com

Group type: Organisation

32 members

Apply to join this group

Danil Ishkov

Ukraine

No Biography provided

You may also be interested in...

Driver to hide files in Linux OS

#COBOLrocks TechCasts: New tricks for COBOL devs

Simple WDM LoopBack Driver

Visual COBOL New Release: Small point. Big deal

Driver Development Part 1: Introduction to Drivers

A Lap Around @ChakraCore

Comments and Discussions

17 messages have been posted for this article Visit **http://www.codeproject.com/Articles/112474/A-Simple-Driver-for-Linux-OS** to post and view comments on this article, or click **here** to get a print view with messages.