Лабораторная работа №8. Взаимодействие с драйверами устройств

Взаимодействие с модулями ядра — драйверами

Если требуется обеспечить настройку и работу устройства, необходимо не только написать драйвер устройства — модуль ядра — но и обеспечить возможность настраивать это устройство через него.

Интересным и практичным примером может также служить случай, когда необходимо получать данные с физических адресов данных, а не с виртуальных. Например, если микропроцессорная система имеет доступ к системной памяти устройства, в которое она встроена. Разумеется, это можно проводить только в пространстве ядра.

Для задания поддерживаемых драйвером операций нужно описать экземпляр структуры file_operations.

```
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(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
         ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
         ssize_t(*aio_write) (struct kiocb *, const char __user *, 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 (*aio_fsync) (struct kiocb *, 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 *);
         ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                             void __user *);
         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);
};
```

Основными функциями часто выступают: открытие, закрытие, чтение, запись.

Взаимодействие через файл устройства

Зарегистрировав драйвер в системе и получив для него старший номер, можно создавать файлы устройств, которые будут работать под управлением этого

драйвера.

Так, записывая что-то в файл (например, echo “command value” > /dev/chardev) или копируя из него, можно взаимодействовать с драйвером. Для этого он должен проводить анализ и определять, какие команды ему передаются.

Пример подобного драйвера приведён ниже.

```
/*
 *  chardev.c: Creates a read-only char device that says how many times
 *  you've read from the dev file
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>        /* for put_user */

/*
 *  Prototypes - this would normally go in a .h file
 */
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev"   /* Dev name as it appears in /proc/devices   */
#define BUF_LEN 80              /* Max length of the message from the device */

/*
 * Global variables are declared as static, so are global within the file.
 */

static int Major;               /* Major number assigned to our device driver
*/
static int Device_Open = 0;     /* Is device open?
                                 * Used to prevent multiple access to device */
static char msg[BUF_LEN];       /* The msg the device will give when asked */
static char *msg_Ptr;

static struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};

/*
 * This function is called when the module is loaded
 */
int init_module(void)
{
        Major = register_chrdev(0, DEVICE_NAME, &fops);

        if (Major < 0) {
          printk(KERN_ALERT "Registering char device failed with %d\n", Major);
          return Major;
        }

        printk(KERN_INFO "I was assigned major number %d. To talk to\n",
Major);
```

```
        printk(KERN_INFO "the driver, create a dev file with\n");
        printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
        printk(KERN_INFO "Try various minor numbers. Try to cat and echo
to\n");
        printk(KERN_INFO "the device file.\n");
        printk(KERN_INFO "Remove the device file and module when done.\n");

        return SUCCESS;
}

/*
 * This function is called when the module is unloaded
 */
void cleanup_module(void)
{
        /*
         * Unregister the device
         */
        int ret = unregister_chrdev(Major, DEVICE_NAME);
        if (ret < 0)
                printk(KERN_ALERT "Error in unregister_chrdev: %d\n", ret);
}

/*
 * Methods
 */

/*
 * Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
        static int counter = 0;

        if (Device_Open)
                return -EBUSY;

        Device_Open++;
        sprintf(msg, "I already told you %d times Hello world!\n", counter++);
        msg_Ptr = msg;
        try_module_get(THIS_MODULE);

        return SUCCESS;
}

/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
        Device_Open--;          /* We're now ready for our next caller */

        /*
         * Decrement the usage count, or else once you opened the file, you'll
         * never get get rid of the module.
         */
        module_put(THIS_MODULE);

        return 0;
}

/*
```

```
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h   */
                           char *buffer,        /* buffer to fill with data */
                           size_t length,       /* length of the buffer     */
                           loff_t * offset)
{
        /*
         * Number of bytes actually written to the buffer
         */
        int bytes_read = 0;

        /*
         * If we're at the end of the message,
         * return 0 signifying end of file
         */
        if (*msg_Ptr == 0)
                return 0;

        /*
         * Actually put the data into the buffer
         */
        while (length && *msg_Ptr) {

                /*
                 * The buffer is in the user data segment, not the kernel
                 * segment so "*" assignment won't work.  We have to use
                 * put_user which copies data from the kernel data segment to
                 * the user data segment.
                 */
                put_user(*(msg_Ptr++), buffer++);

                length--;
                bytes_read++;
        }

        /*
         * Most read functions return the number of bytes put into the buffer
         */
        return bytes_read;
}

/*
 * Called when a process writes to dev file: echo "hi" > /dev/hello
 */
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
        printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
        return -EINVAL;
}
```

Взаимодействие через процесс

Если необходимо обеспечивать контроль ввода/вывода устройства через процесс, то создаётся набор специальных функций IOCTL (Input Output Control). Благодаря им можно более тонко управлять процессом ввода и вывода, например, читая только какой-то определённый байт из устройства.

Также это позволяет взаимодействовать с драйвером из любого процесса, а не через

командную строку.

Простейший пример подобной организации ввода/вывода приведён ниже. Заголовочный файл нужно включать как в исходный код самого модуля, так и в исходный код процесса.

```
/*
 *  chardev.c - Create an input/output character device
 */

#include <linux/kernel.h>       /* We're doing kernel work */
#include <linux/module.h>       /* Specifically, a module */
#include <linux/fs.h>
#include <asm/uaccess.h>        /* for get_user and put_user */

#include "chardev.h"
#define SUCCESS 0
#define DEVICE_NAME "char_dev"
#define BUF_LEN 80

/*
 * Is the device open right now? Used to prevent
 * concurent access into the same device
 */
static int Device_Open = 0;

/*
 * The message the device will give when asked
 */
static char Message[BUF_LEN];

/*
 * How far did the process reading the message get?
 * Useful if the message is larger than the size of the
 * buffer we get to fill in device_read.
 */
static char *Message_Ptr;

/*
 * This is called whenever a process attempts to open the device file
 */
static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
        printk(KERN_INFO "device_open(%p)\n", file);
#endif

        /*
         * We don't want to talk to two processes at the same time
         */
        if (Device_Open)
                return -EBUSY;

        Device_Open++;
        /*
         * Initialize the message
         */
        Message_Ptr = Message;
        try_module_get(THIS_MODULE);
        return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
```

```
#ifdef DEBUG
        printk(KERN_INFO "device_release(%p,%p)\n", inode, file);
#endif

        /*
         * We're now ready for our next caller
         */
        Device_Open--;

        module_put(THIS_MODULE);
        return SUCCESS;
}

/*
 * This function is called whenever a process which has already opened the
 * device file attempts to read from it.
 */
static ssize_t device_read(struct file *file,   /* see include/linux/fs.h   */
                           char __user * buffer,        /* buffer to be
                                                         * filled with data */
                           size_t length,       /* length of the buffer     */
                           loff_t * offset)
{
        /*
         * Number of bytes actually written to the buffer
         */
        int bytes_read = 0;

#ifdef DEBUG
        printk(KERN_INFO "device_read(%p,%p,%d)\n", file, buffer, length);
#endif

        /*
         * If we're at the end of the message, return 0
         * (which signifies end of file)
         */
        if (*Message_Ptr == 0)
                return 0;

        /*
         * Actually put the data into the buffer
         */
        while (length && *Message_Ptr) {

                /*
                 * Because the buffer is in the user data segment,
                 * not the kernel data segment, assignment wouldn't
                 * work. Instead, we have to use put_user which
                 * copies data from the kernel data segment to the
                 * user data segment.
                 */
                put_user(*(Message_Ptr++), buffer++);
                length--;
                bytes_read++;
        }

#ifdef DEBUG
        printk(KERN_INFO "Read %d bytes, %d left\n", bytes_read, length);
#endif

        /*
         * Read functions are supposed to return the number
         * of bytes actually inserted into the buffer
```

```
         */
        return bytes_read;
}

/*
 * This function is called when somebody tries to
 * write into our device file.
 */
static ssize_t
device_write(struct file *file,
             const char __user * buffer, size_t length, loff_t * offset)
{
        int i;

#ifdef DEBUG
        printk(KERN_INFO "device_write(%p,%s,%d)", file, buffer, length);
#endif

        for (i = 0; i < length && i < BUF_LEN; i++)
                get_user(Message[i], buffer + i);

        Message_Ptr = Message;

        /*
         * Again, return the number of input characters used
         */
        return i;
}

/*
 * This function is called whenever a process tries to do an ioctl on our
 * device file. We get two extra parameters (additional to the inode and file
 * structures, which all device functions get): the number of the ioctl called
 * and the parameter given to the ioctl function.
 *
 * If the ioctl is write or read/write (meaning output is returned to the
 * calling process), the ioctl call returns the output of this function.
 *
 */
int device_ioctl(struct inode *inode,   /* see include/linux/fs.h */
                 struct file *file,     /* ditto */
                 unsigned int ioctl_num,        /* number and param for ioctl
*/
                 unsigned long ioctl_param)
{
        int i;
        char *temp;
        char ch;

        /*
         * Switch according to the ioctl called
         */
        switch (ioctl_num) {
        case IOCTL_SET_MSG:
                /*
                 * Receive a pointer to a message (in user space) and set that
                 * to be the device's message.  Get the parameter given to
                 * ioctl by the process.
                 */
                temp = (char *)ioctl_param;

                /*
                 * Find the length of the message
```

```
                 */
                get_user(ch, temp);
                for (i = 0; ch && i < BUF_LEN; i++, temp++)
                        get_user(ch, temp);

                device_write(file, (char *)ioctl_param, i, 0);
                break;

        case IOCTL_GET_MSG:
                /*
                 * Give the current message to the calling process -
                 * the parameter we got is a pointer, fill it.
                 */
                i = device_read(file, (char *)ioctl_param, 99, 0);

                /*
                 * Put a zero at the end of the buffer, so it will be
                 * properly terminated
                 */
                put_user('\0', (char *)ioctl_param + i);
                break;

        case IOCTL_GET_NTH_BYTE:
                /*
                 * This ioctl is both input (ioctl_param) and
                 * output (the return value of this function)
                 */
                return Message[ioctl_param];
                break;
        }

        return SUCCESS;
}

/* Module Declarations */

/*
 * This structure will hold the functions to be called
 * when a process does something to the device we
 * created. Since a pointer to this structure is kept in
 * the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions.
 */
struct file_operations Fops = {
        .read = device_read,
        .write = device_write,
        .ioctl = device_ioctl,
        .open = device_open,
        .release = device_release,      /* a.k.a. close */
};

/*
 * Initialize the module - Register the character device
 */
int init_module()
{
        int ret_val;
        /*
         * Register the character device (atleast try)
         */
        ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);

        /*
```

```
         * Negative values signify an error
         */
        if (ret_val < 0) {
                printk(KERN_ALERT "%s failed with %d\n",
                       "Sorry, registering the character device ", ret_val);
                return ret_val;
        }

        printk(KERN_INFO "%s The major device number is %d.\n",
               "Registeration is a success", MAJOR_NUM);
        printk(KERN_INFO "If you want to talk to the device driver,\n");
        printk(KERN_INFO "you'll have to create a device file. \n");
        printk(KERN_INFO "We suggest you use:\n");
        printk(KERN_INFO "mknod %s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);
        printk(KERN_INFO "The device file name is important, because\n");
        printk(KERN_INFO "the ioctl program assumes that's the\n");
        printk(KERN_INFO "file you'll use.\n");

        return 0;
}

/*
 * Cleanup - unregister the appropriate file from /proc
 */
void cleanup_module()
{
        int ret;

        /*
         * Unregister the device
         */
        ret = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

        /*
         * If there's an error, report it
         */
        if (ret < 0)
                printk(KERN_ALERT "Error: unregister_chrdev: %d\n", ret);
}
```

```
/*
 *  chardev.h - the header file with the ioctl definitions.
 *
 *  The declarations here have to be in a header file, because
 *  they need to be known both to the kernel module
 *  (in chardev.c) and the process calling ioctl (ioctl.c)
 */

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/*
 * The major device number. We can't rely on dynamic
 * registration any more, because ioctls need to know
 * it.
 */
#define MAJOR_NUM 100

/*
 * Set the message of the device driver
```

```
 */
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/*
 * _IOR means that we're creating an ioctl command
 * number for passing information from a user process
 * to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device
 * number we're using.
 *
 * The second argument is the number of the command
 * (there could be several with different meanings).
 *
 * The third argument is the type we want to get from
 * the process to the kernel.
 */

/*
 * Get the message of the device driver
 */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/*
 * This IOCTL is used for output, to get the message
 * of the device driver. However, we still need the
 * buffer to place the message in to be input,
 * as it is allocated by the process.
 */

/*
 * Get the n'th byte of the message
 */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/*
 * The IOCTL is used for both input and output. It
 * receives from the user a number, n, and returns
 * Message[n].
 */

/*
 * The name of the device file
 */
#define DEVICE_FILE_NAME "char_dev"

#endif
```

```
/*
 *  ioctl.c - the process to use ioctl's to control the kernel module
 *
 *  Until now we could have used cat for input and output.  But now
 *  we need to do ioctl's, which require writing our own process.
 */

/*
 * device specifics, such as ioctl numbers and the
 * major device file.
 */
#include "chardev.h"

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>              /* open */
```

```
#include <unistd.h>             /* exit */
#include <sys/ioctl.h>          /* ioctl */

/*
 * Functions for the ioctl calls
 */

ioctl_set_msg(int file_desc, char *message)
{
        int ret_val;

        ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

        if (ret_val < 0) {
                printf("ioctl_set_msg failed:%d\n", ret_val);
                exit(-1);
        }
}

ioctl_get_msg(int file_desc)
{
        int ret_val;
        char message[100];

        /*
         * Warning - this is dangerous because we don't tell
         * the kernel how far it's allowed to write, so it
         * might overflow the buffer. In a real production
         * program, we would have used two ioctls - one to tell
         * the kernel the buffer length and another to give
         * it the buffer to fill
         */
        ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

        if (ret_val < 0) {
                printf("ioctl_get_msg failed:%d\n", ret_val);
                exit(-1);
        }

        printf("get_msg message:%s\n", message);
}

ioctl_get_nth_byte(int file_desc)
{
        int i;
        char c;

        printf("get_nth_byte message:");

        i = 0;
        do {
                c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);

                if (c < 0) {
                        printf
                            ("ioctl_get_nth_byte failed at the %d'th byte:\n",
                             i);
                        exit(-1);
                }

                putchar(c);
        } while (c != 0);
        putchar('\n');
```

```
}

/*
 * Main - Call the ioctl functions
 */
main()
{
        int file_desc, ret_val;
        char *msg = "Message passed by ioctl\n";

        file_desc = open(DEVICE_FILE_NAME, 0);
        if (file_desc < 0) {
                printf("Can't open device file: %s\n", DEVICE_FILE_NAME);
                exit(-1);
        }

        ioctl_get_nth_byte(file_desc);
        ioctl_get_msg(file_desc);
        ioctl_set_msg(file_desc, msg);

        close(file_desc);
}
```

Задание

1. Изучите приведённые примеры взаимодействия с драйверами устройств.

2. Выполните одно из следующих заданий (вариант взять у преподавателя). Требуется реализовать драйвер, поддерживающий функции открытия, закрытия, записи, чтения и имеющий документацию. При записи в драйвер могут передаваться команды. Для этого нужно проводить общий анализ передаваемых в него строк, чтобы определять какие команды передаются. Одна из них direction [forward/back] – направление дальнейшего чтения из драйвера. Например последовательность команд с драйвером

   echo “direction back” > /dev/chardev

   cp /dev/chardev text

   приведёт к чтению строки или буфера драйвера в обратном порядке, если драйвер поддерживает работу со строками. То же для файлов.

   a) Драйвер поддерживает чтение и запись сообщений в него через существующие утилиты POSIX:

   echo “message text_message” > /dev/chardev

   Хранит только одно сообщение. Поддерживает функцию удаления сообщения: msg_delete.

   b) Драйвер поддерживает чтение и запись сообщений в него через пользовательский процесс: ioctl_set_msg(file_desc, msg). Поддерживает функцию удаления сообщения: ioctl_msg_delete.

   c) Драйвер поддерживает чтение и запись файлов в него через

существующие утилиты POSIX:

cat file > /dev/chardev

Хранит только один файл. Поддерживает функцию удаления файла: file_delete.

d) Драйвер поддерживает чтение и запись файла в него через пользовательский процесс: ioctl_write_file(file_desc, file_write_desc). Поддерживает функцию удаления сообщения: ioctl_file_delete.

e) Драйвер поддерживает чтение файла в формате ASCII и запись сообщений в него через существующие утилиты POSIX:

echo “message text_message” > /dev/chardev

Хранит только одно сообщение. Поддерживает функцию удаления сообщения: msg_delete.

f) Драйвер поддерживает чтение файла в формате ASCII и запись сообщений в него через пользовательский процесс: ioctl_set_msg(file_desc, msg). Поддерживает функцию удаления сообщения: ioctl_msg_delete.

Исходный код должен быть реализован в нескольких файлах. Следует обязательно разделить: документацию драйвера, функции инициализации и закрытия, функцию чтения, функцию записи, анализ команд.

Представить отчёт по лабораторной работе, содержащий задание, метод выполнения, возникшие сложности и пути их решения. Сделать собственные выводы о пользе или неэффективности реализованного механизма взаимодействия с драйвером.