Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе**

**Дисциплина**: Проектирование ОС и компонентов

**Тема**: Разработка драйвера символьного устройства под ОС Linux

Выполнил студент гр. 13541/4 Степанов Д.С.

(подпись)

Руководитель Душутина Е.В.

(подпись)

“ ” 2017 г.

Санкт - Петербург

2017

**Цель работы**

Разработать драйвер для символьного устройства в ОС Linux, продемонстрировать его работоспособность. В качестве символьного устройства создадим свое устройство. Задача драйвера будет считывать записанные в устройство символы и записывать их в файл лога. Работоспособность драйвера будет проверяться на Linux debian 3.16.0-4-586

**Выполнение работы**

Имеются два главных пути для общения модуля с процессами. Первый идет через файлы устройства (подобно файлам в каталоге /dev), другой должен использовать файловую систему proc. Первоначальная цель файлов устройства состоит в том, чтобы позволить процессам связываться с драйверами устройства в ядре, и через них с физическими устройствами (модемы, терминалы, и т.д.).

Каждый драйвер устройства, который является ответственным за некоторый тип аппаратных средств, имеет собственный главный номер. Список драйверов и их главных номеров доступен в /proc/devices. Каждое физическое устройство, управляемое драйвером устройства имеет собственный номер. Каталог /dev включает специальный файл, названный файлом устройства, для каждого из тех устройств, которые реально установлены в системе.

Устройства разделены на два типа: символьные и блочные. Различие в том, что блочные имеют буфер для запросов, так что они могут выбирать в каком порядке отвечать. Это важно в случае устройств памяти, где скорее понадобится читать или писать сектора, которые ближе друг к другу, чем те, которые находятся далеко. Другое различие: блочные устройства могут принимать ввод и возвращать вывод только в блоках (чей размер может измениться согласно устройству), в то время как символьные устройства могут использовать столько байтов, сколько нужно. Большинство устройств в мире символьные, потому что они не нуждаются в этом типе буферизации и не работают с фиксированным размером блока. Для того чтобы узнать, является ли устройство блочным или символьным, можно обратить внимание на первый символ в выводе ls -l. Если это "b", значит устройство блочное, а если "c", то символьное.

Драйвер символьного устройства является модулем ядра, следовательно, в нем обязательно должны быть определены функции загрузки модуля в ядро (module\_init()) и выгрузки модуля из ядра (module\_exit()). Для того, что бы ядро знала операции, которые драйвер будет выполнять, они прописываются в структуре file\_operations, определенной в заголовочном файле linux/fs.h:

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 (\*aio\_read) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);

ssize\_t (\*aio\_write) (struct kiocb \*, const struct iovec \*, unsigned long, 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 \*\*);

long (\*fallocate)(struct file \*file, int mode, loff\_t offset,

loff\_t len);

int (\*show\_fdinfo)(struct seq\_file \*m, struct file \*f);

};

Данная структура содержит указатели на функции драйвера, которые отвечают за выполнение различных операций с устройством. Драйвер зачастую реализует далеко не все функции, предусмотренные данной структурой. Поля структуры, соответствующие нереализованным функциям, заполняются ”пустыми” указателями - NULL. Для простейшего драйвера реализуем 2 функции:

* open - открытие устройства;
* release - закрытие устройства.

Исходный код драйвера приведен ниже:

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <asm/uaccess.h>

// Ниже мы задаём информацию о модуле, которую можно будет увидеть с помощью Modinfo

MODULE\_AUTHOR( "Player1" );

MODULE\_DESCRIPTION( "Lab3 module" );

MODULE\_SUPPORTED\_DEVICE( "lab3" ); /\* /dev/testdevice \*/

#define DEVICE\_NAME "lab3"

// Поддерживаемые нашим устройством операции

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 \* );

// Глобальные переменные, объявлены как static, воизбежание конфликтов имен.

static int major\_number; /\* Старший номер устройства нашего драйвера \*/

static int is\_device\_open = 0; /\* Используется ли девайс ? \*/

static char text[ 5 ] = "test\n"; /\* Текст, который мы будет отдавать при обращении к нашему устройству \*/

static char\* text\_ptr = text; /\* Указатель на текущую позицию в тексте \*/

// Прописываем обработчики операций на устройством

static struct file\_operations fops =

{

.read = device\_read,

.write = device\_write,

.open = device\_open,

.release = device\_release

};

// Функция загрузки модуля. Входная точка.

static int \_\_init test\_init( void )

{

printk( KERN\_ALERT "TEST driver loaded!\n" );

// Регистрируем устройсво и получаем старший номер устройства

major\_number = register\_chrdev( 0, DEVICE\_NAME, &fops );

if ( major\_number < 0 )

{

printk( "Registering the character device failed with %d\n", major\_number );

return major\_number;

}

// Сообщаем присвоенный нам старший номер устройства

printk( "Test module is loaded!\n" );

return 0;

}

// Функция выгрузки модуля

static void \_\_exit test\_exit( void )

{

// Освобождаем устройство

unregister\_chrdev( major\_number, DEVICE\_NAME );

printk( KERN\_ALERT "Test module is unloaded!\n" );

}

// Указываем наши функции загрузки и выгрузки

module\_init( test\_init );

module\_exit( test\_exit );

static int device\_open( struct inode \*inode, struct file \*file )

{

text\_ptr = text;

if ( is\_device\_open )

return -EBUSY;

is\_device\_open++;

return 0;

}

static int device\_release( struct inode \*inode, struct file \*file )

{

is\_device\_open--;

return 0;

}

device\_write( struct file \*filp, const char \*buff, size\_t len, loff\_t \* off )

{

printk( "Write in device\n" );

return -EINVAL;

}

static ssize\_t device\_read( struct file \*filp, /\* include/linux/fs.h \*/

char \*buffer, /\* buffer \*/

size\_t length, /\* buffer length \*/

loff\_t \* offset )

{

int byte\_read = 0;

if ( \*text\_ptr == 0 )

return 0;

while ( length && \*text\_ptr )

{

put\_user( \*( text\_ptr++ ), buffer++ );

length--;

byte\_read++;

}

return byte\_read;

}

Для компиляции драйвера необходимо написать makefile:

obj-m += driver.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

Для встраивания модулей в ядро воспользуемся уже знакомой утилитой insmod. Также с помощью утилиты modinfo можно посмотреть информацию о драйвере:

root@debian:/home/user/SystemProg/Sem2/lab3# /sbin/modinfo driver.ko

filename: /home/user/SystemProg/Sem2/lab3/driver.ko

description: Lab3 module

author: Player1

depends:

vermagic: 3.16.0-4-586

Поэкспериментируем с созданным драйвером:

root@debian:/home/user/SystemProg/Sem2/lab3# insmod driver.ko

root@debian:/home/user/SystemProg/Sem2/lab3# dmesg | tail -n 2

[ 4636.944855] TEST driver loaded!

[ 4636.944862] Test module is loaded!

root@debian:/home/user/SystemProg/Sem2/lab3# mknod /dev/driver c 250 0

root@debian:/home/user/SystemProg/Sem2/lab3# cat /dev/driver

test

root@debian:/home/user/SystemProg/Sem2/lab3# echo 1 > /dev/driver

bash: echo: ошибка записи: Недопустимый аргумент

root@debian:/home/user/SystemProg/Sem2/lab3# rmmod driver

root@debian:/home/user/SystemProg/Sem2/lab3# dmesg | tail -n 4

[ 4636.944855] TEST driver loaded!

[ 4636.944862] Test module is loaded!

[ 4703.926104] Write in device

[ 4736.694199] Test module is unloaded!

**Разработка драйвера клавиатуры**

Создадим драйвер, который будет перехватывать прерывание нажатия клавиш и записывать нажатый код в системный лог. Исходный код драйвера представлен ниже.

#include <linux/module.h>

#include <linux/interrupt.h>

#include <linux/init.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <asm/io.h>

#include <asm/irq\_vectors.h>

// Функция получения кода нажатой клавиши

irq\_handler\_t irq\_handler (int irq, void \*dev\_id, struct pt\_regs \*regs)

{

static unsigned char scancode;

// Прерывание для получения считанного кода

scancode = inb(0x60);

// Вывод считанной клавиши

printk(KERN\_INFO "Scan Code %u\n",(unsigned int)scancode);

return (irq\_handler\_t) IRQ\_HANDLED;

}

// Инициализация модуля

int init\_module ()

{

// Запрос прерывания IRQ 1 (прерывание клавиатуры)

// Необходимо для того, что бы перейти на наш обработчик

return request\_irq (1, (irq\_handler\_t) irq\_handler, IRQF\_SHARED, "test\_keyboard\_irq\_handler", (void \*)(irq\_handler));

}

// Выгрузка модуля

void cleanup\_module()

{

// Освобождаем запрошенное прерывание

free\_irq(1, NULL);

}

Эксперимент:

root@debian:/home/user/SystemProg/Sem2/lab3# insmod mykeyb.ko

root@debian:/home/user/SystemProg/Sem2/lab3# dmesg | tail

[ 462.328330] Scan Code 20

[ 462.349248] Scan Code 185

[ 462.406757] Scan Code 30

[ 462.469704] Scan Code 148

[ 462.486601] Scan Code 23

[ 462.570510] Scan Code 158

[ 462.581039] Scan Code 151

[ 462.662489] Scan Code 38

[ 462.810618] Scan Code 166

[ 463.618996] Scan Code 28

**Вывод**

В данной лабораторной работе были изучены основные принципы разработки драйверов для ОС Linux, а так же создан простой драйвер для символьного устройства. Драйвер в ОС Linux яв- 8 ляется модулем ядра, но при его создании необходимо так же указать, какие функции он будет выполнять (при помощи заполнении структуры file\_operations), а так же задать ему идентифи- катор. Созданный в ходе работы драйвер выполнял только операции чтения и записи, что было достаточно для усвоения основных принципов разработки драйверов.

**Список использованных источников**

1. Интернет источник. The linux documentation project <http://www.tldp.org/LDP/lkmpg/2.6/html/x245.html> Дата обращения: 18.04.2017
2. Linux Device Drivers, Third Edition by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman 2005
3. Интернет источник. https://www.opennet.ru/docs/RUS/lkmpg/node14.html «Файлы символьных устройств» Дата обращения: 18.04.2017