**Тема**: Создание драйвера символьного устройства в ОС Linux

**Цель работы**

Разработать драйвер символьного устройства для ОС Linux. В качестве символьного устройства создать свое устройство. Драйвер должен работать в качестве посредника между программой-клиентом и реальным символьным устройством. Его задача в том, чтобы передавать все запросы от клиента реальному устройству и ответы устройства обратно клиенту. Так же при это он должен записывать все сообщения в системный лог.

**Выполнение работы**

Специальный файл устройства, или просто файл устройства (англ. special device file) — это один из типов файлов в UNIX-подобной операционной системе. Специальные файлы устройств содержат данные, необходимые операционной системе для взаимодействия с физическими устройствами, такими как диски и дисководы, принтеры и факсы и т. п. Фактически, специальные файлы устройств являются указателями на драйверы устройств, и когда процесс обращается к файлу устройств, он по сути работает с драйвером этого устройства.

Поскольку в операционной системе есть разные типы устройств, то и файлы устройств бывают разными. Есть два типа файлов устройств: блочные (англ. block special files) и символьные (англ. character special files). Блочные файлы устройств используются для передачи данных, разделённых на пакеты фиксированной длины — блоки. А символьные файлы устройств используются для небуферизованного обмена данными. Большинство устройств способно принимать и отправлять данные либо блоками (блочные устройства), либо сплошным потоком байтов (символьные устройства), но некоторые (такие как жёсткий диск) сочетают в себе обе эти возможности. Работа с первым типом устройств возможна либо через блочные, либо через символьные файлы, а вот с последним типом — подходят и те, и другие.

Для того, чтобы операционная система могла определить файл устройства и получить некоторые характеристики о самом устройстве, в файле содержатся 3 специальных поля: класс устройства, старший номер устройства и младший номер устройства. Класс устройства сообщает символьное устройство или блочное. В некоторых операционных системах (например, в Linux) есть и дополнительный класс устройств — небуферизованное символьное устройство. Кроме класса, есть ещё и тип устройства, который можно узнать по старшему номеру, например, в Linux системе 1 означает оперативную память, 2 — дисковод гибких дисков, 3 — первый контроллер для жестких IDE-дисков, и т. д. В разных операционных системах один и тот же старший номер может указывать на разные типы устройств. Для того, чтобы отличить два устройства одного класса и типа предусмотрели младший номер. Он используется для нумерации устройств с одинаковыми старшими номерами

Специальные файлы устройств в Linux расположены в каталоге /dev:

kivi@kivi-VirtualBox:~/Phone$ ls -l /dev/

total 0

crw------- 1 root root 10, 235 апр 16 11:46 autofs

drwxr-xr-x 2 root root 620 апр 16 11:46 block

drwxr-xr-x 2 root root 80 апр 16 11:46 bsg

crw------- 1 root root 10, 234 апр 16 11:46 btrfs-control

drwxr-xr-x 3 root root 60 апр 16 11:46 bus

lrwxrwxrwx 1 root root 3 апр 16 11:46 cdrom -> sr0

drwxr-xr-x 2 root root 3580 апр 16 11:47 char

crw------- 1 root root 5, 1 апр 16 11:46 console

lrwxrwxrwx 1 root root 11 апр 16 11:46 core -> /proc/kcore

drwxr-xr-x 2 root root 60 апр 16 11:46 cpu

crw------- 1 root root 10, 59 апр 16 11:46 cpu\_dma\_latency

crw------- 1 root root 10, 203 апр 16 11:46 cuse

drwxr-xr-x 5 root root 100 апр 16 11:46 disk

drwxr-xr-x 2 root root 80 апр 16 11:46 dri

lrwxrwxrwx 1 root root 3 апр 16 11:46 dvd -> sr0

crw------- 1 root root 10, 61 апр 16 11:46 ecryptfs

crw-rw---- 1 root video 29, 0 апр 16 11:46 fb0

……

С символьными устройствами проводилась работа в прошлом семестре при работе с прерываниями от клавиатуры:

#include <linux/module.h>

#include <linux/init.h>

#include <linux/interrupt.h>

#include <linux/stat.h>

#include <asm/io.h>

#include <asm/irq\_vectors.h>

#define SHARED\_IRQ 17

static int irq = SHARED\_IRQ, my\_dev\_id, irq\_counter = 0;

module\_param( irq, int, S\_IRUGO );

static irqreturn\_t my\_interrupt( int irq, void \*dev\_id ) {

static unsigned char scancode;

irq\_counter++;

scancode = inb(0x60);

printk( KERN\_INFO "In the ISR: counter = %d\n", irq\_counter );

printk( KERN\_INFO "Scancode: = %d\n", scancode );

return IRQ\_NONE; /\* we return IRQ\_NONE because we are just observing \*/

}

static int \_\_init my\_init( void ) {

if ( request\_irq( irq, my\_interrupt, IRQF\_SHARED, "my\_interrupt", &my\_dev\_id ) )

return -1;

printk( KERN\_INFO "Successfully loading ISR handler on IRQ %d\n", irq );

return 0;

}

static void \_\_exit my\_exit( void ) {

synchronize\_irq( irq );

free\_irq( irq, &my\_dev\_id );

printk( KERN\_INFO "Successfully unloading, irq\_counter = %d\n", irq\_counter );

}

module\_init( my\_init );

module\_exit( my\_exit );

MODULE\_LICENSE( "GPL v2" );

Данный модуль ядра встраивается в систему и перехватывает прерывания от клавиатуры. В модуле мы определили три функции. Функция my\_interrupt – наш обработчик прерывания. Эта функция просто считает количество поступивших прерываний, считывает просканированный код и печатает это число в буфер сообщений ядра.

Функция my\_init – функция инициализации, которая вызывается при загрузке модуля в ядро. В этой функции мы пытаемся установить наш обработчик прерывания с помощью системного вызова request\_irq. При успешной установке эта функция печатает соответствующее сообщение в буфер сообщений.

Функция my\_exit – функция завершения, которая вызывается при выгрузке модуля из ядра. В ней мы ожидаем освобождения обработчика (synchronize\_irq), снимаем свой обработчик (free\_irq) и печатаем соответствующее сообщение.

Makefile для компиляции модуля:

obj-m += irq.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

Проверим работу модуля:

kivi@kivi-VirtualBox:~/sp\_labs/irq\_exc/irq$ sudo insmod irq.ko irq=1

kivi@kivi-VirtualBox:~/sp\_labs/irq\_exc/irq$ test

kivi@kivi-VirtualBox:~/sp\_labs/irq\_exc/irq$ sudo rmmod irq

kivi@kivi-VirtualBox:~/sp\_labs/irq\_exc/irq$ dmesg | tail -n 10

[ 575.955134] Scancode: = 19

[ 576.041535] In the ISR: counter = 33

[ 576.041550] Scancode: = 15

[ 576.063177] In the ISR: counter = 34

[ 576.063184] Scancode: = 147

[ 576.168596] In the ISR: counter = 35

[ 576.168611] Scancode: = 143

[ 576.413462] In the ISR: counter = 36

[ 576.413468] Scancode: = 28

[ 576.419068] Successfully unloading, irq\_counter = 36

**Реализация собственного драйвера**

Драйвер символьного устройства реализуется в виде модуля ядра, соответственно в нем должны быть определены функции module\_init() и module\_exit(). Так же, в модуле должен быть создан экземпляр структуры file\_operations, который определяет функции, которые будут применимы к созданному файлу устройства.

Данная структура объявлена в 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 (\*read\_iter) (struct kiocb \*, struct iov\_iter \*);

ssize\_t (\*write\_iter) (struct kiocb \*, struct iov\_iter \*);

int (\*iterate) (struct file \*, struct dir\_context \*);

int (\*iterate\_shared) (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

ssize\_t (\*copy\_file\_range)(struct file \*, loff\_t, struct file \*,

loff\_t, size\_t, unsigned int);

int (\*clone\_file\_range)(struct file \*, loff\_t, struct file \*, loff\_t,

u64);

ssize\_t (\*dedupe\_file\_range)(struct file \*, u64, u64, struct file \*,

u64);

};

Перейдем к реализации самого модуля. Вначале объявляем все необходимые переменные и функции:

#define DEVICE\_NAME "sniffer"

#define CLASS\_NAME "sniff"

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Azat Abdullin");

MODULE\_DESCRIPTION("A simple Linux char driver sniffer");

MODULE\_VERSION("0.1");

static int majorNumber; // старший номер устройства

static struct class\* snifferClass = NULL; // класс устройства

static struct device\* snifferDevice = NULL; // устройство

static struct file\* fd; // файловый дескриптор файла реального устройтсва

static mm\_segment\_t old\_fs; // сегмент VFS для операций ввода/вывода

static char\* real\_device = "/dev/ttyUSB0"; // путь к файлу реального устройства

// Прототипы функций операций над файлом

static int dev\_open(struct inode \*, struct file \*);

static int dev\_release(struct inode \*, struct file \*);

static ssize\_t dev\_read(struct file \*, char \*, size\_t, loff\_t \*);

static ssize\_t dev\_write(struct file \*, const char \*, size\_t, loff\_t \*);

// Структура с определением допустимых операций

static struct file\_operations fops = {

.open = dev\_open,

.read = dev\_read,

.write = dev\_write,

.release = dev\_release,

};

Мы определили для файла 4 операции: открытие, закрытие, чтение и запись.

Функции инициализации и выгрузки модуля ядра:

static int \_\_init sniffer\_init(void) {

printk(KERN\_INFO "sniffer: Initializing the sniffer LKM\n");

// Try to dynamically allocate a major number for the device

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

if (majorNumber<0) {

printk(KERN\_ALERT "sniffer failed to register a major number\n");

return majorNumber;

}

printk(KERN\_INFO "sniffer: registered correctly with major number %d\n", majorNumber);

// Register the device class

snifferClass = class\_create(THIS\_MODULE, CLASS\_NAME);

if (IS\_ERR(snifferClass)) {

unregister\_chrdev(majorNumber, DEVICE\_NAME);

printk(KERN\_ALERT "Failed to register device class\n");

return PTR\_ERR(snifferClass);

}

printk(KERN\_INFO "sniffer: device class registered correctly\n");

// Register the device driver

snifferDevice = device\_create(snifferClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE\_NAME);

if (IS\_ERR(snifferDevice)) {

class\_destroy(snifferClass);

unregister\_chrdev(majorNumber, DEVICE\_NAME);

printk(KERN\_ALERT "Failed to create the device\n");

return PTR\_ERR(snifferDevice);

}

printk(KERN\_INFO "sniffer: device class created correctly\n");

return 0;

}

static void \_\_exit sniffer\_exit(void) {

device\_destroy(snifferClass, MKDEV(majorNumber, 0)); // remove the device

class\_unregister(snifferClass); // unregister the device class

class\_destroy(snifferClass); // remove the device class

unregister\_chrdev(majorNumber, DEVICE\_NAME); // unregister the major number

printk(KERN\_INFO "sniffer: Goodbye from the LKM!\n");

}

module\_init(sniffer\_init);

module\_exit(sniffer\_exit);

Сначала динамически регистрируется старший номер устройства с помощью функции register\_chrdev(). В нее передаются в качестве аргументов:

* Старший номер устройства (или 0 для динамического выделения номера)
* Имя устройства
* Указатель на структуру file\_operations для устройства

Далее создается структура класса устройства с помощью функции class\_create(). В качестве аргументов ей передаются указатель на текущий модуль и имя устройства.

Далее создается специальный файл устройства с помощью функции device\_create(). Ее аргументы:

* Указатель на структуру класса устройства
* Указатель на структуру родительского устройства (если оно есть; в нашем случае NULL)
* Тип создаваемого устройства. Заполняется с помощью макроса MKDEV, который объединяет старший и младший номер устройства в необходимый тип
* Имя создаваемого устройства

В функции выгрузки модуля сначала с помощью функции device\_destroy() удаляется файл устройства, затем выгружается и удаляется структура класса устройства (с помощью функций class\_unregister() и class\_destroy()). Далее удаляется старший номер устройства с помощью функции unregister\_chrdev().

Так как задача нашего устройства – перенаправлять все сообщения реальному устройству, в функциях доступа к файлу будет осуществляться ввод/вывод со специальным файлом реального устройства. Так как в модулях ядра нельзя использовать стандартные функции файловых операций (open, close, read, write), были написаны собственные реализации этих функций, которые работают на уровне виртуальной файловой системы:

struct file\* file\_open(const char\* path, int flags, int rights) {

struct file\* filp = NULL;

mm\_segment\_t oldfs;

int err = 0;

oldfs = get\_fs();

set\_fs(get\_ds());

filp = filp\_open(path, flags, rights);

set\_fs(oldfs);

if(IS\_ERR(filp)) {

err = PTR\_ERR(filp);

return NULL;

}

return filp;

}

void file\_close(struct file\* file) {

filp\_close(file, NULL);

}

int file\_read(struct file\* file, unsigned long long offset, unsigned char\* data, unsigned int size) {

mm\_segment\_t oldfs;

int ret;

oldfs = get\_fs();

set\_fs(get\_ds());

ret = vfs\_read(file, data, size, &offset);

set\_fs(oldfs);

return ret;

}

int file\_write(struct file\* file, unsigned long long offset, const unsigned char\* data, unsigned int size) {

mm\_segment\_t oldfs;

int ret;

oldfs = get\_fs();

set\_fs(get\_ds());

ret = vfs\_write(file, data, size, &offset);

set\_fs(oldfs);

return ret;

}

Далее опишем функции доступа к файлу устройства:

static int dev\_open(struct inode \*inodep, struct file \*filep) {

printk(KERN\_INFO "sniffer: Device has been opened \n");

/// Connecting to real device

fd = file\_open (real\_device, O\_RDWR | O\_NOCTTY | O\_SYNC, 0);

if (fd == NULL) {

printk(KERN\_INFO "sniffer: cannot open device %s\n", real\_device);

return 1;

}

return 0;

}

static ssize\_t dev\_read(struct file \*filep, char \*buffer, size\_t len, loff\_t \*offset) {

int error\_count = 0;

int i = 0;

error\_count = file\_read(fd, \*offset, buffer, len);

if (error\_count > 0) {

printk(KERN\_INFO "sniffer: readed %d bytes\n", error\_count);

for (i = 0; i < error\_count; ++i) {

printk(KERN\_INFO "sniffer: %c", buffer[i]);

}

printk(KERN\_INFO "sniffer: message end\n");

}

return error\_count;

}

static ssize\_t dev\_write(struct file \*filep, const char \*buffer, size\_t len, loff\_t \*offset) {

file\_write(fd, \*offset, buffer, len);

printk(KERN\_INFO "sniffer: Message to device: %s\n", buffer);

return len;

}

static int dev\_release(struct inode \*inodep, struct file \*filep) {

printk(KERN\_INFO "sniffer: Device successfully closed\n");

file\_close(fd);

return 0;

}

Данные функции просто перенаправляют обращения к нашему устройству на реальное устройство и записывают все действия в системный лог.

Makefile для модуля:

obj-m += sniffer.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 проверки работоспособности драйвера была написана небольшая программа, которая выполняет к нему обращения:

#include <string.h>

#include <unistd.h>

#include <stdio.h>

int main(int argc, char\*\* argv) {

char \*portname = "/dev/sniffer";

if (argc > 1)

portname = argv[1];

int fd = open (portname, O\_RDWR | O\_NOCTTY | O\_SYNC);

if (fd < 0) {

printf("error opening the device\n");

return 1;

}

write (fd, "at+cgmm\n", 8);

usleep ((8 + 25) \* 100);

char buf [100];

int n = read (fd, buf, sizeof buf);

printf("%s\n", buf);

return 0;

}

Данная программа просто открывает файл драйвера, пишет в него сообщение, ждет небольшое время пока сообщение отправится и затем считывает, и выводит ответ от устройства.

Проверим модуль, когда у нас не подключено реальное устройство:

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$ make

make -C /lib/modules/4.7.5-custom/build M=/home/kivi/Phone/sniffer\_driver/driver modules

make[1]: Entering directory '/home/kivi/Downloads/linux-4.7.5'

CC [M] /home/kivi/Phone/sniffer\_driver/driver/sniffer.o

/home/kivi/Phone/sniffer\_driver/driver/sniffer.c:40:21: warning: ‘old\_fs’ defined but not used [-Wunused-variable]

static mm\_segment\_t old\_fs;

^

Building modules, stage 2.

MODPOST 1 modules

CC /home/kivi/Phone/sniffer\_driver/driver/sniffer.mod.o

LD [M] /home/kivi/Phone/sniffer\_driver/driver/sniffer.ko

make[1]: Leaving directory '/home/kivi/Downloads/linux-4.7.5'

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$ sudo insmod sniffer.ko

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$ ls -l /dev/sniffer

crw------- 1 root root 246, 0 апр 16 12:39 /dev/sniffer

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$ cd ../client/

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/client$ make

cc -DDISPLAY\_STRING -o dev\_test dev\_test.c

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/client$ sudo ./dev\_test /dev/sniffer

error opening the device

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/client$ cd ../driver/

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$ sudo rmmod sniffer

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$ dmesg | grep sniffer

[ 2871.346607] sniffer: Initializing the sniffer LKM

[ 2871.346611] sniffer: registered correctly with major number 246

[ 2871.347040] sniffer: device class registered correctly

[ 2871.348376] sniffer: device class created correctly

[ 2881.124663] sniffer: Device has been opened

[ 2881.124668] sniffer: cannot open device /dev/ttyUSB0

[ 2881.124674] Modules linked in: sniffer(OE) vboxsf(OE) intel\_powerclamp vboxvideo(OE) snd\_intel8x0 crct10dif\_pclmul crc32\_pclmul snd\_ac97\_codec ghash\_clmulni\_intel joydev ac97\_bus aesni\_intel vboxguest(OE) aes\_x86\_64 snd\_pcm lrw gf128mul snd\_seq\_midi glue\_helper snd\_seq\_midi\_event snd\_rawmidi ablk\_helper ttm drm\_kms\_helper cryptd snd\_seq snd\_seq\_device drm snd\_timer fb\_sys\_fops snd i2c\_piix4 input\_leds syscopyarea soundcore serio\_raw sysfillrect sysimgblt mac\_hid binfmt\_misc parport\_pc ppdev lp parport autofs4 hid\_generic usbhid hid e1000 fjes video ahci libahci psmouse pata\_acpi

[ 2888.056623] sniffer: Goodbye from the LKM!

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$

Как видим, модуль был успешно загружен и инициализирован. Однако, при попытке обратиться к устройству возникли ошибка, т.к. драйвер не смог обратиться к реальному устройству.

Подключим реальное устройство и проверим работу драйвера:

// команда at – проверка состояния устройства

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/client$ sudo ./dev\_test /dev/sniffer

OK

// команда at+cgmm – информация о идентификации продукта

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/client$ sudo ./dev\_test /dev/sniffer

SIMCOM\_SIM808

OK

// команда atd - звонок

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/client$ sudo ./dev\_test /dev/sniffer

BUSY

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$ sudo insmod sniffer.ko

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$ sudo rmmod sniffer

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$ dmesg | grep sniffer

[ 282.052519] sniffer: Initializing the sniffer LKM

[ 282.052523] sniffer: registered correctly with major number 246

[ 282.052531] sniffer: device class registered correctly

[ 282.054170] sniffer: device class created correctly

[ 363.734696] sniffer: Device has been opened

[ 363.785041] sniffer: Message to device: at

[ 363.790881] sniffer: readed 6 bytes

[ 363.790884] sniffer:

[ 363.790885] sniffer:

[ 363.790886] sniffer: O

[ 363.790887] sniffer: K

[ 363.790888] sniffer:

[ 363.790888] sniffer:

[ 363.790889] sniffer: message end

[ 797.281139] sniffer: Device successfully closed

[ 310.476898] sniffer: Device has been opened

[ 310.530069] sniffer: Message to device: at+cgmm

[ 310.539092] sniffer: readed 23 bytes

[ 310.539095] sniffer:

[ 310.539096] sniffer:

[ 310.539097] sniffer: S

[ 310.539098] sniffer: I

[ 310.539099] sniffer: M

[ 310.539099] sniffer: C

[ 310.539100] sniffer: O

[ 310.539100] sniffer: M

[ 310.539101] sniffer: \_

[ 310.539102] sniffer: S

[ 310.539102] sniffer: I

[ 310.539103] sniffer: M

[ 310.539104] sniffer: 8

[ 310.539104] sniffer: 0

[ 310.539105] sniffer: 8

[ 310.539105] sniffer:

[ 310.539106] sniffer:

[ 310.539107] sniffer:

[ 310.539108] sniffer:

[ 310.539108] sniffer: O

[ 310.539109] sniffer: K

[ 310.539110] sniffer:

[ 310.539110] sniffer:

[ 310.539111] sniffer: message end

[ 310.539227] sniffer: Device successfully closed

[ 780.710827] sniffer: Device has been opened

[ 780.770578] sniffer: Message to device: atd89110124987;

[ 797.280805] sniffer: readed 8 bytes

[ 797.280812] sniffer:

[ 797.280816] sniffer:

[ 797.280819] sniffer: B

[ 797.280821] sniffer: U

[ 797.280824] sniffer: S

[ 797.280826] sniffer: Y

[ 797.280828] sniffer:

[ 797.280830] sniffer:

[ 797.280832] sniffer: message end

[ 363.791007] sniffer: Device successfully closed

[ 386.234976] sniffer: Goodbye from the LKM!

kivi@kivi-VirtualBox:~/Phone/sniffer\_driver/driver$

По результатам видно, что написанный драйвер позволяет обращаться к используемому устройству и предоставляет нам всю его функциональность. Так же в логе ядра мы можем увидеть все сообщения, которые были переданы драйверу.

**Вывод**

В данной работе был изучен процесс разработки драйверов для символьного устройства в ОС Linux. В качестве примера был реализован простейший модуль ядра, который перехватывает прерывания от клавиатуры, которая является символьным устройством, и считывает просканированный код.

Так же был создан простой драйвер символьного устройства, который пересылает все запросы к нему на реальное устройство и логирует их. Для указания доступных функций драйвера необходимо создать структуру file\_operations и передать в нее указатели на функции реализуемых операций. Так же необходимо указать в драйвере старший и младший номер устройства и зарегистрировать его в системе. В данной работе были реализованы операции открытия/закрытия и чтения/записи для специального файла устройства. Написанный драйвер был протестирован и была показана его работоспособность.

**Список использованных источников**

1. Linux kernel: <https://www.kernel.org/>