Страница для тайтла

Страничка для ТЗ

Содержание

Введение 4

1 Аналитический раздел 5
1.1 Процесс обработки прерываний 5
1.1.1 Контроллер прерываний 5
1.1.2 Обработчик прерываний 6
1.2 Отложенное действие 6
1.3 Процедуры доступа к портам ввода/вывода 7
1.4 Драйвер символьных устройств 8
1.4.1 Старший и младший номера устройств 9
1.4.2 Структуры данных символьного устройства 9
1.4.3 Регистрация символьных устройств 11
1.4.4 Вывод 11
1.5 Подсистема ввода ядра 11

2 Конструкторискый раздел 14
2.1 Реализация модуля ядра 14
2.2 Основнеые структуры программы 15
2.3 Формат конфигурации от пользователей 15
2.4 Реализация обработчика прерываний 16

3 Технологический раздел 17
3.1 Выбор инструментария 17
3.1.1 Язые программирования 17
3.1.2 Среда разработки 17
3.2 Реализация 17
3.2.1 Makefile 18
3.2.2 Загружаемый модуль ядра 18

Заключение 20

Список литературы 21

Приложение 22

Введение

На сегодняшний день существует множество причин по которым клавиатура должна быть сконфигурирована для использования в качестве мыши. Одной из них является распространенное использование беспроводных компьютерных мышей с батарейным питанием, которые любят иссекать свой запас в самый неподходящий момент, да и могут просто выйти из строя. Так же немаловажным является, то что людям с проблемой мобильности рук намного проще нажимать клавиши на клавиатуре, чем двигать рукой по столу.

Существует несколько путей достижения данной функциональности в операционой системе (ОС) Linux, но достаточно часто они очень сложны для рядового пользователя или лишены некоторых ключевых функций.

Среди существующих аналогов можно выделить следующие:

1) xbindkeys - работает, но настройка окажется сложной для рядового пользователя;

2) MouseKeys - поддерживает только Ubuntu и требует наличие цифровой клавиатуры.

Практической целью курсовой работы является разработка загружаемого модуля ядра, который позволит пользователю легко управлять курсором мышь с помощью клавиатуры и настраивать конфигурацию.

В соответствии с заданием на курсовой проект необходимо разработать модуль ядра для управления курсором мыши с помощью клавиатуры.

Для решения поставленной цели необходимо выполнить следующие задачи:

1) проанализировать процесс обработки прерываний;

2) проанализировать процедуры доступа к портам ввода/вывода;

3) проанализировать драйвер символьных устройств;

4) проанализировать подсистему ввода ядра;

5) разработать модуль ядра для управления курсором мыши.

1 Аналитический раздел

В данном разделе рассмотрены: процесс обработки прерываний; процедуры доступа к портам ввода/вывода; драйвер символьного устройстваю; подсистема ввода ядра.

1.1 Процесс обработки прерываний

Прерывание - это сообщение, информирующее систему о том, что одно из устройств выполнило операцию или на нем произошла ошибка. Прерывание заставляет процессор приостановить выполнение программы и вызвать операционную систему, чтобы иметь возможность ответить на прерывание [3]. Прерывания могут быть сгруперованы в две категории в зависимости от источника прерывания:

1) синхронные прерывания или внутренние (исключения) - генерируются при выполнении иснтрукции. Они обрабатывают условия, обнаруженные процессором при выполнении инструкции;

2) асинхронные прерывания - являются классическим типом прерываний и вызываются периферийными устройствами в произвольное время. В отличие от синхронных прерываний, асинхронные не связаны каким-нибудь процессом. Они возникают в любое время независимо от состояния системы и легко выполнимы.

1.1.1 Контроллер прерываний

Устройство, поддерживающее прерывания, имеет выходной контакт используемый для отправки запроса прерывания (IRQ). Каждый из этих контактов называется линией прерывания и подключен к устройству под названием Контролер прерывайний (Programmable Interrupt Controllet, PIC), которое подключено к контакту intr процессора. Схема контролера прерываний показана на рисунке 1.

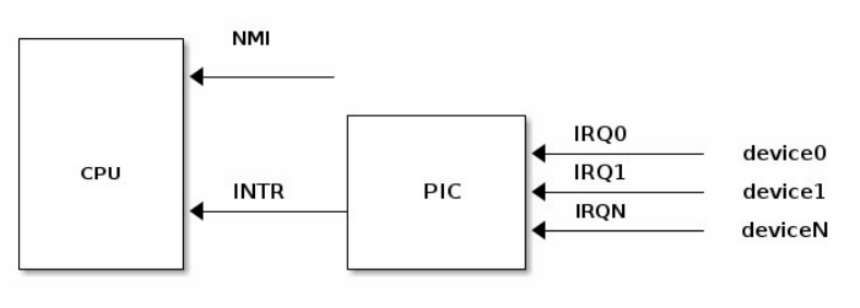

Рис. 1. Схема контролера прерываний

1.1.2 Обработчик прерываний

Как и с другими ресурсами, перед тем как и их использовать, модуль запрашивает канал прерывания (IRQ) и так же особождает его, когда заканчивает работу. Во многих случиях ожидается, что модули будут способны делить линии прерывания с другими драйверами.

В Linux запрос на получение и особождение прерывания выполняется с помощью функций request_irq() и free_irq объявленных в <linux/interrupt.h> (листинг 1)

Листинг 1. Функция request_irq() и free_irq()

```
1 int request_irq(unnsigned int irq_no,
2                 irqreturn_t (*handler)(int, void *, struct pt_regs *),
3                 unsigned long flags,
4                 const char *dev_name,
5                 void *dev_id);
6
7 void free_irq(unsigned int irq_no, void *dev_id);
```

После того как прерывание было запрошено, оно будет обработано функцией обработчика.

1.2 Отложенное действие

Отложенное действие - это класс средст ядра, который позволяет планировать выполнение кода на более позднее время. Оно используется для дополнения функциональности обработчика прерываний. При использовании отложенного действия минимальная требуемая работа выполняется в обра-

ботчике прерываний, а остальные операции будут запланированы из обработчика прерываний для выполнения позже.

В настоящее время существует 3 механизма отложенного действия:

1) softirqs - могут использоваться драйверами устройств и зарезервированы для подсистем ядра;

2) тасклеты - они работают в контексте прерывания, как и softirq. Основное отличие заключается в том, что тасклеты могут выделяться динамически и, следовательно, использоваться драйверами устройств;

3) очереди работ - используются для планирования действия, выполняемых в контексте процесса.

В работе планируется использовать IRQ1 (контролер клавиатуры) для захвата прерываний с клавиатуры

В данной работе будет будем использовать тасклет, поскольку он работает относительно быстрее очереди работ. Так же он может быть использован в драйвере устройства в отличии от softirq

1.3 Процедуры доступа к портам ввода/вывода

В Linux доступ к портам ввода-вывода реализован на всех архитектурах, и существует несколько API, которые можно использовать.

Перед доступом к портам ввода-вывода необходимо вызвать запрос доступа, чтобы убедиться, что существует только один пользователь. Для этого используется функция request_region(). Чтобы освободить зарезервированную область, необходимо использовать функцию release_region().

После получения нужного порта ввода-вывода на нем можно выполнять операции чтения или записи. Поскольку физические порты различаются по количеству битов (8, 16 или 32 бита), существуют различные функции доступа к портам в зависимости от их размера. В asm/io.h определены следующие функции доступа к портам:

1) unsigned inb(int port): чтение одного байта из порта;

2) void outb(unsigned char byte, int port): запись одного байта в порт;

3) unsigned inw(int port): чтение двух байтов из порта;

4) void outw(unsigned short word, int port): запись двух байтов в порт;

5) unsigned inl(int port): чтение четырех байтов из порта;

6) void outl(unsigned long word, int port): запись четырех байтов в порт.

1.4 Драйвер символьных устройств

В UNIX доступ к аппаратным устройствам осуществляется пользователем через специальные файлы устройств. Эти файлы группируются в каталог /dev, а системные вызовы open, read, write, close, и т. д. перенаправляются операционной системой на драйвер устройства, связанный с физическим устройством. Драйвер устройства – это компонент ядра (обычно модуль), которая предназначена для управления конкретным устройством. Обычно драйверы устройств содержат последовательность команд, специфичных для конкретного устройства. Поскольку драйвер предназначен управления устройством, то код должен соответствовать специфике устройства. Обычно это связано с форматом передачи данных от системы к устройству и обратно.

В системах UNIX все устройства разделены на два типа [5]:

1) блочные - блочное устройство хранит данные и производит ввод-вывод блоками фиксированного размера, доступными в произвольном порядке. Обычно размер блока равняется 512 байтам, умноженным на 2 в степени, где степень больше либо равно 0. В качестве примеров блочных устройств можно указать жесткие диски, привод компакт-дисков [5];

2) символьные - символьные устройства могут использоватсья для хранения и передачи данных произвольного объема. Некоторые устройства этого типа умеют передавать информацию побайтно, вырабатывая каждый раз прерывание. Данные устройства не в состоянии использовать произвольную адрисацию и не поддерживают операцию поиска. Примерами устройств такого типа являются терминалы, принтеры, "мыши"и звуковые карты [5].

1.4.1 Старший и младший номера устройств

Индетификация и обращение к устройствам определяется пространством имен устройств. В системе Unix существует три различных пространста имен устройств:

1) аппаратное пространство;

2) ядро;

3) пользовательское.

Ядро индетифицирует устройство по типу (блочное или символьное), а также по паре номеров, получивших название старжего и младшего номера устройств (major или minor). Старший номер устройства идинтифицирует его драйвер. Младший номер устройства идентифицирует определенный экземпляр устройства [5].

1.4.2 Структуры данных символьного устройства

В ядре устройство символьного типа представлено структурой struct cdev, используемой для его регистрации в системе. Большинство операций драйвера используют три важные структуры:

1) struct file_operations;

2) struct file;

3) struct inode;

Драйверы символьных устройств получают системные вызовы, выполняемые пользователями через файлы типа устройств. Другими словами, реализация драйвера символьного устройства означает реализацию системных вызовов, специфичных для файлов: open, close, read,write и т. д. Эти операции описаны в листинге 2 структуры struct file_operations [6].

Листинг 2. Структура struct file_operations

```
1 #inlcude <linux/fs.h>
2
```

```
3  struct file_operations {
4      struct module *owner;
5      loff_t (*llseek) (struct file *, loff_t, int);
6      ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
7      ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
8      ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
9      ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
10     int (*iopoll)(struct kiocb *kiocb, bool spin);
11     int (*iterate) (struct file *, struct dir_context *);
12     int (*iterate_shared) (struct file *, struct dir_context *);
13     __poll_t (*poll) (struct file *, struct poll_table_struct *);
14     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
15     long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
16     int (*mmap) (struct file *, struct vm_area_struct *);
17     unsigned long mmap_supported_flags;
18     int (*open) (struct inode *, struct file *);
19     int (*flush) (struct file *, fl_owner_t id);
20     int (*release) (struct inode *, struct file *);
21     int (*fsync) (struct file *, loff_t, loff_t, int datasync);
22     int (*fasync) (int, struct file *, int);
23     int (*lock) (struct file *, int, struct file_lock *);
24     ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
           int);
25     unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
           long, unsigned long, unsigned long);
26     int (*check_flags)(int);
27     int (*flock) (struct file *, int, struct file_lock *);
28     ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t
           *, size_t, unsigned int);
29     ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
           size_t, unsigned int);
30     int (*setlease)(struct file *, long, struct file_lock **, void **);
31     long (*fallocate)(struct file *file, int mode, loff_t offset,
32     loff_t len);
33     void (*show_fdinfo)(struct seq_file *m, struct file *f);
34     #ifndef CONFIG_MMU
35     unsigned (*mmap_capabilities)(struct file *);
36     #endif
37     ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
38     loff_t, size_t, unsigned int);
39     loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
40     struct file *file_out, loff_t pos_out,
41     loff_t len, unsigned int remap_flags);
42     int (*fadvise)(struct file *, loff_t, loff_t, int);
43 }
```

1.4.3 Регистрация символьных устройств

Регистрация или отмена регистрации устройства производится путем указания старшего и млашего номера устройств. Тип dev_t используется для хранения идентификаторов устройсва и может быть получен с помощью макроса MKDEV

Идентификаторы устройств могут быть статически назначены с помощью register_chrdev_region() или динамически распределены alloc_chrdev_region(). После вывода символьного устройства необходимо не забыть вызвать unregister_chrdev_region() для освобожднеия распредеения [7].

После присвоение идентификатора, символьное устройство должно быть иницилизировано cdev_init, а ядро должно быть уведомлено cdev_add. Функция cdev_add вызывается только после того, как устройство будет готово к приему вызовов. Удаление устройства производится с помощью функции cdev_del [7].

1.4.4 Вывод

Для данного проекта был выбран драйвер символьного устройство из за небольшого объема данных. Индентификатор устройства будет выделен статически, если заданы страшний и млаший номер. Если данные номера отсутствуют, то они будут распределены динамически.

1.5 Подсистема ввода ядра

Подсистема ввода была введена для унификации различных драйверов, управляющих различными устройствами, такими как компьютерные мыши, клавиатуры, сенсорные экраны и т.п. Подсистема ввода дает различные преимущества:

1) единая обработка функционально похожих устройств, даже если они конструктивно разные. Например: USB и Bluetooth мыши обрабатываютсяв системе одинаково;

2) Простой событийный интерфейс для отправки пользовательского ввода приложениям. драйверу не приходится создавать и управлять узлом в каталоге /dev. Вместо этого он может использовать API для отображения изменения положения мыши или события нажатия одной из клавиш.

Подсистема содержит два класса драйверов: драйверы событий и драйверы устройств. Драйверы событий отвечают за взаимодействие с приложениями, тогда как драйверы устройств отвечают за низкоуровневую связь с устройствами ввода. Рисунок 2 иллюстрирует работу подсистемы ввода.

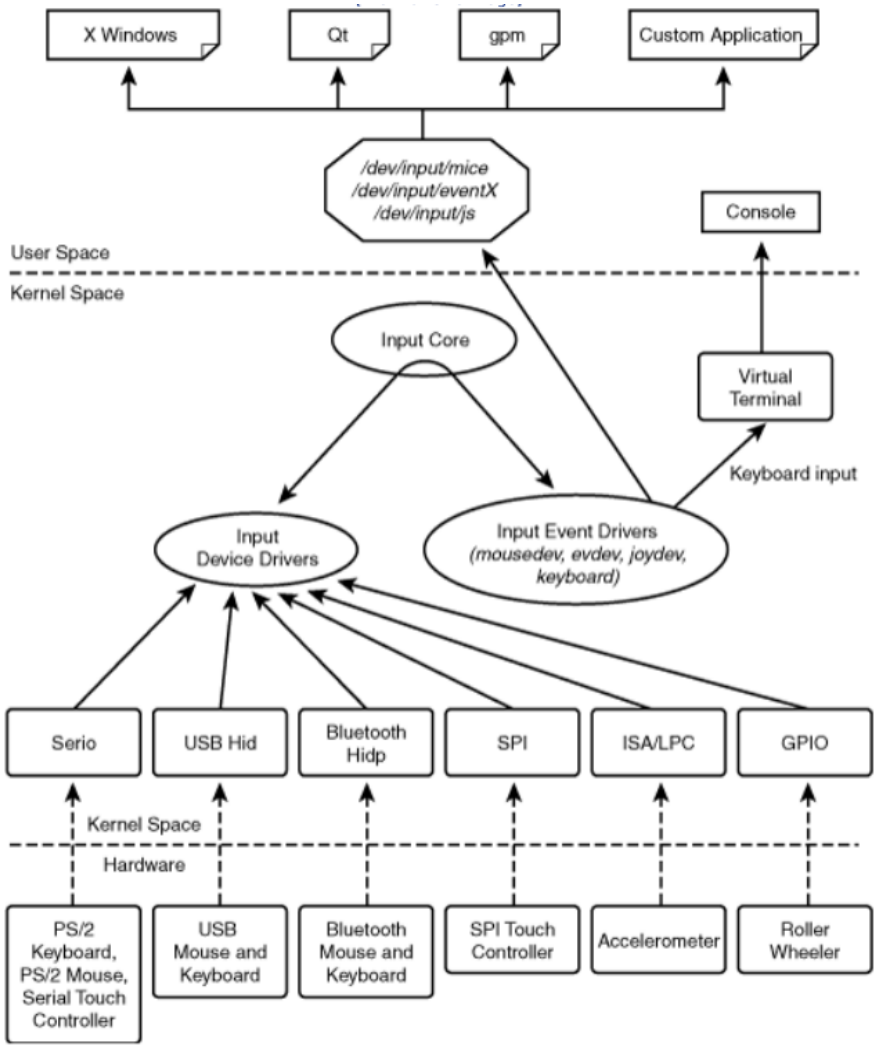

Рис. 2. Структура подсистемы ввода

Драйверы событий предлагают аппаратно независимую абстракцию для взаимодействия с устройствами ввода. evdev (устройство события) - это общий интерфейс ввода событий в ядре Linux. Он обобщает необработанные события от драйверов устройств и делает их доступными через символьные устройства в каталоге /dev/input/.

Каждое событие имеет структуру показаную в листинге 3.

Листинг 3. Структура события evdev

```
struct input_event {
    struct timeval time;     // Timestamp
    __u16 type;              // Event type
    __u16 code;              // Event code
    __s32 value;             // Event value
}
```

Основные типы событий испускаемые evdev:

1) EV_SYN - разделение событий;

2) EV_KEY - для отображения нажатия клавиш клавиатуры, мыши или других кнопочных устройств;

3) EV_REL - передача относительного изменения координат, например при движении компьютерной мышью.

2 Конструкторискиий раздел

В данном разделе рассписаны: реализация модуля ядра, основные структуры программы, формат конфигарции приложения от пользователя, реализация обработчика прерываний.

2.1 Реализация модуля ядра

1) Создать драйвер символьного устройства для чтения конфигурации из пользовательского пространства и захвата прерывания клавиатуры;

2) Запросить порты ввода-вывода клавиатуры;

3) Зарегистрировать обработчик IRQ для прерывания клавиатуры. Он будет захватывать все прерывания клавиатуры и хранить scancode в буфере. Затем буфер будет обработан тасклетом;

4) Создать устройство мыши для работы с устройством;

5) Зарегистрировать устройство мыши в подсистеме ввода ядра;

6) Запускать соответствующих событий устройства мыши при распознавании правильных комбинаций клавиш в тасклете.

На рисунке 3 представелна структура системы

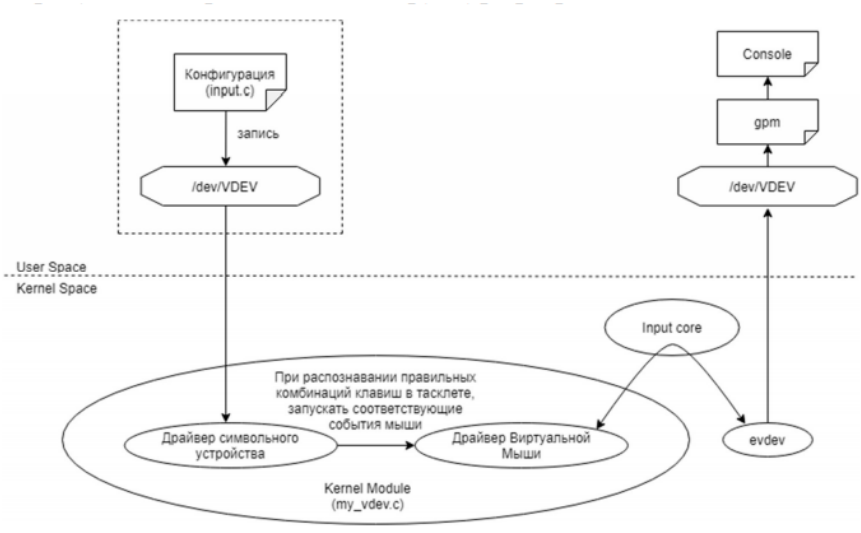

Рис. 3.

2.2 Основнеые структуры программы

В этой курсовой работе символьное устройство обернуто другой структурой, называемой vdev. Эта структура действует на некоторую память, выделенную из ядра и содержит буфер scancode для символьного устройства. Она также содержит конфигурацию пользователя (листинг 4).

Листинг 4. Структура vdev

```
static struct vdev {
    struct cdev cdev;
    spinlock_t lock;
    u8 buf[2];
    char map[8];
    int spd;
}
```

Структура struct file_operations определяет, какие операции могут быть выполнены на символьном устройстве. В нашем случаи символьному устройству нужно только получать конфуигурацию из пользовательского пространства, по этому реализуются только операции: открытия, закрытия, чтения (листинг 5).

Листинг 5. Структура file_operations

```
static const struct file_operations vdev_fops = {
    .owner = THIS_MODULE,
    .open = vdev_open,
    .releae = vdev_release,
    .write = vdev_write,
};
```

2.3 Формат конфигурации от пользователей

Для обеспечения пользователей возможность настройки параметров драйвера необходимо установить формат команд. Команда должна состоять издвух частей:

1) код команды:

   а) 0 - настройка карты клавиатуры;

б) 1 - настройка скорости перемещения курсора мыши.

2) тело:

а) если код команды 0, то тело представляет собой символьную строку состаяющую из 6 символов, обозначающих 6 клавиш, соответстующих движению вверх, вниз, влево, вправо, лкм, пкм;

б) если код 1, то тело представляет собой целове цисло, указывающее скорость движения мыши.

2.4 Реализация обработчика прерываний

На рисунке 4 отображена реализация обработчика прерываний клавиатуры.

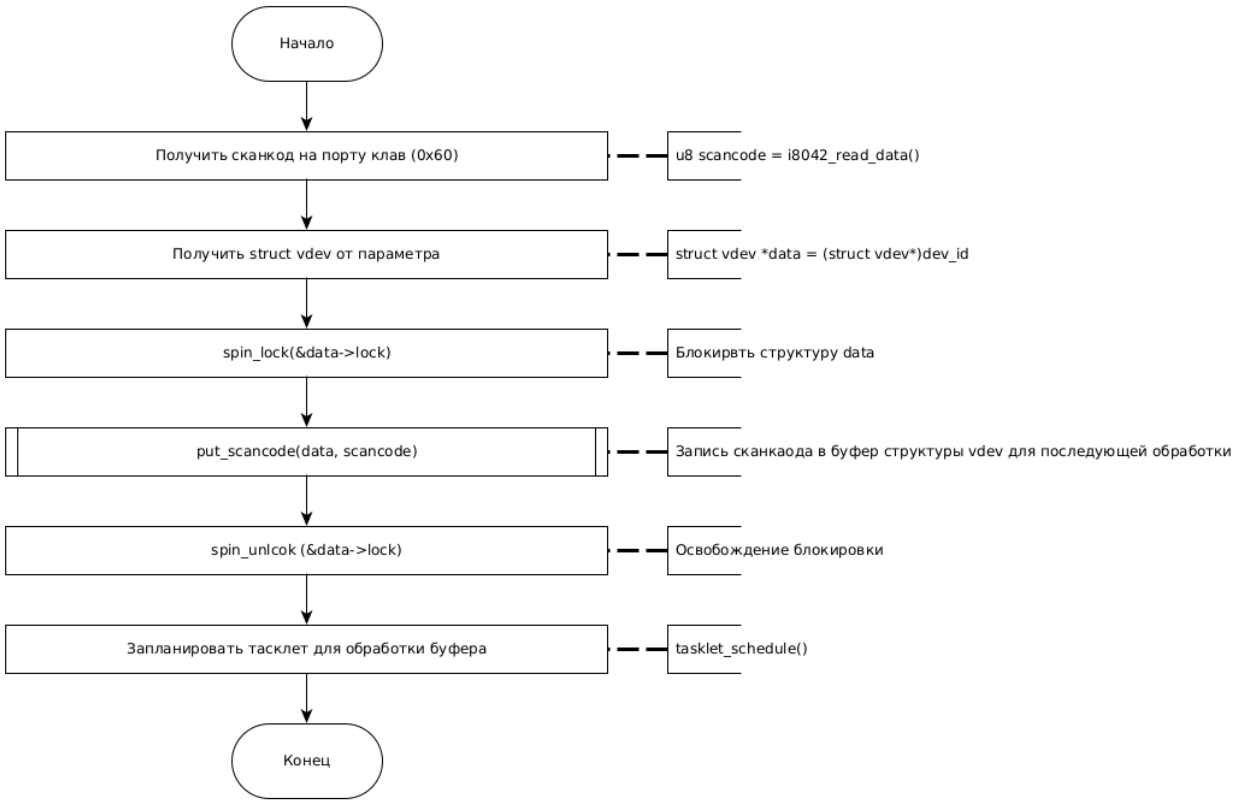

Рис. 4. Реализация обработчика прерываний квалиатуры

3 Технологический раздел

В данном разделе обоснован выбор инструментария. Рассписана реализация модуля ядра.

3.1 Выбор инструментария

3.1.1 Язые программирования

Был выбран язык C стандарта C11 для реализации программы, потому что:

1) ядро написано на языке C. Что означает, что C полностью поддерживает все API ядра;

2) на данный язык наибольшее количество литературы;

3.1.2 Среда разработки

В качестве среды разработки мной был выбран Visual Studio Code.

Visual Studio Code - редактор исходного кода, разработанный Microsoft для Windows, Linux и macOS. Позиционируется как «лёгкий» редактор кода для кроссплатформенной разработки веб- и облачных приложений. Включает в себя отладчик, инструменты для работы с Git, подсветку синтаксиса, IntelliSense и средства для рефакторинга. Имеет широкие возможности для кастомизации: пользовательские темы, сочетания клавиш и файлы конфигурации. Распространяется бесплатно, разрабатывается как программное обеспечение с открытым исходным кодом, но готовые сборки распространяются под проприетарной лицензией.

3.2 Реализация

В данном подразделе рассмотрен makefile и загружаемый модуль ядра.

3.2.1 Makefile

Для облегчения изборки и избежания поторения одних и тех же команд был написан Makefile, который позволяет запускать сборку одной командой.

В листинге 6 представлено содержмое makefile.

Листинг 6. Makefile

```
1 CFLAGS=-Wall
2
3 test: test.o
4
5 .PHONY: clean
6
7 clean:
8 -rm -f *~ *.o
```

Обратите внимание, что все команды должны выполняться к правами администратора или выполняться с командой sudo.

Для сборки модуля ядра необходимо выполнить команду make. Для загрузки данного модуля в ядро, выполнять команду insdmod my_vdev.ko. Для выгрузки модуля из ядра, выполнять команду rmmod my_vdev.

3.2.2 Загружаемый модуль ядра

Важная особенность реализации подсистемы устройств в ОС Linux: драйвер оперирует с устройством как с парой номеров major/minor, а все команды GNU и функции POSIX API оперируют с устройством как с именем в каталоге /dev. Для работы с устройством мы должны установить взаимно однозначное соответствие между major/minor номерами и именем устройства. Для того чтобы задать такое соответствие используется макрос MKDEV.

Система сама может найти подходящий Старший номер, если вызывать функцию alloc_chardev_region.

Для инициализации структуры вызывается функция cdev_init, которая содержит поле const struct file operations *ops, которое определяет доступные операции для работы с файлом.

Функция cdev_add добавляет в структуру cdev зарегистрированный драйвер, структуру dev_t.

Вызов фунции input_allocate_device инстанцирует структуру input_dev, в которой указывается какие события будут переданы устройству: EV_REL, EV_KEY, с помощью метода set_bit.

При записи в файл, данные будут копироваться из из пользовательского пространства в пространство ядра с помощью вызова copy_from_user. Затем новые данные будут сохранены в struct my_vdev.

Реализация обработчика прерываний на основе приведенного выше алгоритма показана в листинге 7

Листинг 7. Листинг тестовой программы

```
irqreturn_t kbd_interrupt_handler(int irq_no, void* dev_id)
{
  u8 scancode = i8042_read_data();

  struct vdev* data = (struct vdev*)dev_id;

  spin_lock(&data->lock);
  put_scancode(data, scancode);
  spin_unlock(&data->lock);

  tasklet_schedule(mouse_tasklet);

  // Report the interrupt as not handled
  // so that the original driver can
  // process it
  return IRQ_NONE;
}
```

В тасклете сканкод в буфере будет проверен во всех случаях, и соответствующее событие мыши будет вызвано с помощью функций input_report_key, input_report_rel.

Заключение

В рамках данного курсового проекта был реализовн модуль ядра под ОС Linux, позволяющий управлять курсором мыши с помощью клавиаутры. В процессе разработки была изучена подсистема ввода ОС Linux и изучены принципы работы с драйверами устройств.

Создаваемый драйвер при дальнейшей разработке может быть изменён следующим образом:

1) модернизация модуля для поддежрки USB-клавиатуры;

2) добавление дополнительных функций в настройки.

Список использованных источников

1. Анатомия загружаемых модулей ядра Linux // URL: *https* : *//www.ibm.com/developerworks/ru/library/l − lkm/index.html* (Дата обращения: 02.11.20)

2. uinput module. //

   URL: *https* : *//www.kernel.org/doc/html/v*4.12/*input/uinput.html* (Дата обращения: 02.11.2020)

3. Х.М. Дейтел, П.ДЖ Дейтел, Д.Р. Чофнес Операционные системы 3е издание ТОМ 1

4. Курс лекций операционных систем Рязанова Н.Ю.

5. Вахалия Ю. UNIX изнутри

6. GitHub Torvalds Linux // URL: *https* : *//github.com/torvalds/linux* (Дата обращения 04.11.2020)

7. J. Corber, A. Rubini, G. Kroah-Hartman Драйверы устройсв Linux, Третье издание

Приложение

Листинг 8. Листинг my_vdev.h

```
1  #ifndef __MY_VDEV_H__
2  #define __MY_VDEV_H__
3
4  #define MODULE_NAME "VDEV"
5
6  #define VDEV_MAJOR 42
7  #define VDEV_MINOR 0
8  #define VDEV_DEV_COUNT 1
9
10 #define I8042_KBD_IRQ 1
11 #define I8042_STATUS_REG 0x64
12 #define I8042_DATA_REG 0x60
13
14 #define SCANCODE_RELEASED_MASK 0x80
15 #define SCANCODE_LALT_MASK 0x38
16
17 #define CMD_MAP 0
18 #define CMD_SPD 1
19
20 #define BUF_SIZE 64
21
22 static struct vdev { // Wrapper struct for char device
23   struct cdev cdev;
24   spinlock_t lock;
25   u8 buf[2]; // buffer to store last 2 pressed key
26   char map[6]; // map for mouse movement: UP, DOWN, LEFT, RIGHT, BTNLEFT,
       BTNRIGHT
27
28   int spd; // mouse movement speed
29 } devs[1];
30 static struct input_dev* mouse_dev;
31 static struct class* dev_class;
32 static struct tasklet_struct* mouse_tasklet;
33
34
35 static inline u8 i8042_read_data(void);          // Return value of data
       register
36 static int is_key_pressed(u8);                   // Check if a given scancode
       corresponds to key press or release
37 static void put_scancode(struct vdev*, u8);      // Put scancode to device data
38 static int scancode_to_ascii(u8);                // Return a character of a
       given scancode
39 void mouse_tasklet_handler(unsigned long);       // Tasklet handler
```

```
40 irqreturn_t kbd_interrupt_handler(int, void*);  // Keyb interrupt handler
41 static int vdev_open(struct inode*, struct file*);
42 static int vdev_release(struct inode*, struct file*);
43
44 // User space -> Device: get config from user
45 static ssize_t vdev_write(struct file*, const char __user*, size_t, loff_t*);
46 #endif
```

Листинг 9. Листинг my_vdev.c

```
1  #include <asm/io.h>
2  #include <linux/cdev.h> // for char device
3  #include <linux/device.h> // for creating device file
4  #include <linux/fs.h>
5  #include <linux/init.h>
6  #include <linux/input.h> // for input device
7  #include <linux/interrupt.h>
8  #include <linux/ioport.h>
9  #include <linux/kdev_t.h> // for creating device file
10 #include <linux/kernel.h>
11 #include <linux/module.h>
12 #include <linux/slab.h> // for kmalloc, kfree
13 #include <linux/spinlock.h>
14 #include <linux/uaccess.h> // for user access
15
16 #include "my_vdev.h"
17
18 MODULE_DESCRIPTION(MODULE_NAME);
19 MODULE_AUTHOR("BroBezzubik");
20 MODULE_LICENSE("GPL");
21
22 static const struct file_operations vdev_fops = {
23    .owner = THIS_MODULE,
24    .open = vdev_open,
25    .release = vdev_release,
26    .write = vdev_write,
27 };
28
29 /********************************* TASKLET
       ***********************************/
30 static int is_key_pressed(u8 scancode)
31 {
32    return !(scancode & SCANCODE_RELEASED_MASK);
33 }
34
35 static int scancode_to_ascii(u8 scancode)
36 {
```

```
37    static char* row1 = "1234567890";
38    static char* row2 = "qwertyuiop";
39    static char* row3 = "asdfghjkl";
40    static char* row4 = "zxcvbnm";
41
42    scancode &= ~SCANCODE_RELEASED_MASK;
43    if (scancode >= 0x02 && scancode <= 0x0b)
44      return *(row1 + scancode - 0x02);
45    if (scancode >= 0x10 && scancode <= 0x19)
46      return *(row2 + scancode - 0x10);
47    if (scancode >= 0x1e && scancode <= 0x26)
48      return *(row3 + scancode - 0x1e);
49    if (scancode >= 0x2c && scancode <= 0x32)
50      return *(row4 + scancode - 0x2c);
51    if (scancode == 0x39)
52      return ' ';
53    if (scancode == 0x1c)
54      return '\n';
55    return '?';
56  }
57
58  void mouse_tasklet_handler(unsigned long arg)
59  {
60    struct vdev* data = (struct vdev*)arg;
61    int pressed;
62
63    pressed = is_key_pressed(data->buf[1]);
64
65    if (pressed) {
66      if (data->buf[0] == SCANCODE_LALT_MASK) {
67        char ch = scancode_to_ascii(data->buf[1]);
68
69        if (ch == data->map[0]) {
70          input_report_rel(mouse_dev, REL_Y, -data->spd);
71          input_sync(mouse_dev);
72        } else if (ch == data->map[1]) {
73          input_report_rel(mouse_dev, REL_Y, data->spd);
74          input_sync(mouse_dev);
75        } else if (ch == data->map[2]) {
76          input_report_rel(mouse_dev, REL_X, -data->spd);
77          input_sync(mouse_dev);
78        } else if (ch == data->map[3]) {
79          input_report_rel(mouse_dev, REL_X, data->spd);
80          input_sync(mouse_dev);
81        } else if (ch == data->map[4]) {
82          input_report_key(mouse_dev, BTN_LEFT, 1);
83          input_sync(mouse_dev);
84        } else if (ch == data->map[5]) {
```

```
85          input_report_key(mouse_dev, BTN_RIGHT, 1);
86          input_sync(mouse_dev);
87        }
88      }
89    } else {
90      if (data->buf[0] == SCANCODE_LALT_MASK) {
91        char ch = scancode_to_ascii(data->buf[1]);
92
93        if (ch == data->map[4]) {
94          input_report_key(mouse_dev, BTN_LEFT, 0);
95          input_sync(mouse_dev);
96        } else if (ch == data->map[5]) {
97          input_report_key(mouse_dev, BTN_RIGHT, 0);
98          input_sync(mouse_dev);
99        }
100     }
101   }
102 }
103
104 // Interupt
105 static inline u8 i8042_read_data(void)
106 {
107   u8 val;
108   val = inb(I8042_DATA_REG);
109   return val;
110 }
111
112 static void put_scancode(struct vdev* data, u8 scancode)
113 {
114   char ch = 0;
115
116   ch = scancode_to_ascii(scancode);
117
118   if (data->buf[0] != SCANCODE_LALT_MASK
119       || (ch == data->map[0] && ch == data->map[1]
120           && ch == data->map[2] && ch == data->map[3])) {
121     data->buf[0] = data->buf[1];
122   }
123
124   data->buf[1] = scancode;
125   //pr_info("VDEV: [0]: 0x%x, [1]: 0x%x", data->buf[0], data->buf[1]);
126 }
127
128 irqreturn_t kbd_interrupt_handler(int irq_no, void* dev_id)
129 {
130   u8 scancode = i8042_read_data();
131
132   struct vdev* data = (struct vdev*)dev_id;
```

```
133
134   spin_lock(&data->lock);
135   put_scancode(data, scancode);
136   spin_unlock(&data->lock);
137
138   tasklet_schedule(mouse_tasklet);
139
140   // Report the interrupt as not handled
141   // so that the original driver can
142   // process it
143   return IRQ_NONE;
144 }
145
146 // Driver functions
147 static int vdev_open(struct inode* inode, struct file* file)
148 {
149   struct vdev* data = container_of(inode->i_cdev, struct vdev, cdev);
150
151   file->private_data = data;
152   pr_info("VDEV: Device file opened\n");
153
154   return 0;
155 }
156
157 static int vdev_release(struct inode* inode, struct file* file)
158 {
159   pr_info("VDEV: Device file closed\n");
160   return 0;
161 }
162
163 static ssize_t vdev_write(struct file* file, const char __user* user_buffer,
164     size_t count, loff_t* offset)
165 {
166   struct vdev* data = (struct vdev*)file->private_data;
167   size_t size = BUF_SIZE < count ? BUF_SIZE : count;
168   char* buf;
169   char cmd;
170
171   if ((buf = (char*)kmalloc(size, GFP_KERNEL)) == NULL) {
172     pr_err("VDEV: kmalloc failed");
173     return -EFAULT;
174   }
175
176   if (copy_from_user(buf, user_buffer, size)) {
177     pr_err("VDEV: copy_from_user failed\n");
178     kfree(buf);
179     return -EFAULT;
180   }
```

```
181
182   // Get cmd from user
183   memcpy(&cmd, buf, sizeof(char));
184   cmd = cmd - '0';
185
186   switch (cmd) {
187   case CMD_MAP:
188     memcpy(&data->map, buf + 2, 6);
189     break;
190   case CMD_SPD:
191     kstrtol(buf + 2, 10, (long int*)&data->spd);
192     break;
193   default:
194     pr_info("VDEV: User config malformed");
195     break;
196   }
197
198   kfree(buf);
199   return size;
200 }
201
202 static int __init vdev_init(void)
203 {
204   int err;
205   dev_t devnum;
206
207   // Register char device
208   if (VDEV_MAJOR) {
209     devnum = MKDEV(VDEV_MAJOR, VDEV_MINOR);
210     err = register_chrdev_region(devnum, VDEV_DEV_COUNT, MODULE_NAME);
211   } else {
212     err = alloc_chrdev_region(&devnum, VDEV_MINOR, VDEV_DEV_COUNT,
          MODULE_NAME);
213   }
214
215   if (err != 0) {
216     pr_err("VDEV: register_region failed: %d\n", err);
217     goto out;
218   }
219
220  // Request I/O ports
221   if (request_region(I8042_DATA_REG + 1, 1, MODULE_NAME) == NULL) {
222     err = -EBUSY;
223     goto out_unregister;
224   }
225   if (request_region(I8042_STATUS_REG + 1, 1, MODULE_NAME) == NULL) {
226     err = -EBUSY;
227     goto out_unregister;
```

```
228   }
229
230   // Spinblock and def confing
231   spin_lock_init(&devs[0].lock);
232   devs[0].map[0] = 'w'; // UP
233   devs[0].map[1] = 's'; // DOWN
234   devs[0].map[2] = 'a'; // LEFT
235   devs[0].map[3] = 'd'; // RIGHT
236   devs[0].map[4] = 'j'; // BTNLEFT
237   devs[0].map[5] = 'k'; // BTNRIGHT
238   devs[0].spd = 10;
239
240   // Register IRQ handler for kb
241   err = request_irq(
242       I8042_KBD_IRQ,           // IRQ line
243       kbd_interrupt_handler,
244       IRQF_SHARED,             // share interrupt line with other vdev driver
              (i8042)
245       MODULE_NAME,             // use this to show dev in /proc/interrupts
246       &devs[0]);               // for share interrupt, dev_id can't be NULL
247   if (err != 0) {
248     pr_err("VDEV: request_irq failed: %d\n", err);
249     goto out_release_regions;
250   }
251
252   // Add char device to system
253   cdev_init(&devs[0].cdev, &vdev_fops);
254   err = cdev_add(&devs[0].cdev, devnum, VDEV_DEV_COUNT);
255   if (err != 0) {
256     pr_err("VDEV: cdev_add failed: %d\n", err);
257     goto out_release_regions;
258   }
259
260   // Create struct class and device file
261   if ((dev_class = class_create(THIS_MODULE, MODULE_NAME)) == NULL) {
262     err = -1;
263     pr_err("VDEV: class_create failed\n");
264     goto out_cdev_del;
265   }
266
267   if ((device_create(dev_class, NULL, devnum, NULL, MODULE_NAME)) == NULL) {
268     err = -1;
269     pr_err("VDEV: device_create failed\n");
270     goto out_class_destroy;
271   }
272
273   // allocate mouse device
274   mouse_dev = input_allocate_device();
```

```
275   if (mouse_dev == NULL) {
276     err = -1;
277     pr_err("VDEV: input_dev registered failed\n");
278     goto out_device_destroy;
279   }
280
281   // Init mouse device
282   mouse_dev->name = MODULE_NAME;
283   mouse_dev->phys = MODULE_NAME;
284   mouse_dev->id.bustype = BUS_VIRTUAL;
285   mouse_dev->id.vendor = 0x0000;
286   mouse_dev->id.product = 0x0000;
287   mouse_dev->id.version = 0x0000;
288
289   set_bit(EV_REL, mouse_dev->evbit);
290   set_bit(REL_X, mouse_dev->relbit);
291   set_bit(REL_Y, mouse_dev->relbit);
292   set_bit(EV_KEY, mouse_dev->evbit);
293   set_bit(BTN_LEFT, mouse_dev->keybit);
294   set_bit(BTN_RIGHT, mouse_dev->keybit);
295
296   // Register mouse device in system
297   err = input_register_device(mouse_dev);
298   if (err != 0) {
299     pr_err("VDEV: input_register_device failed\n");
300     goto out_input_free_device;
301   }
302
303   // Init tasklet mouse
304   if ((mouse_tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL)) ==
        NULL) {
305     err = -1;
306     pr_err("VDEV: kmalloc failed");
307     goto out_input_unregister_device;
308   }
309   tasklet_init(mouse_tasklet, mouse_tasklet_handler, (unsigned long)&devs[0]);
310
311   pr_notice("VDEV: Driver %s loaded\n", MODULE_NAME);
312   return 0;
313
314 // TO GO tags
315 out_input_unregister_device:
316   input_unregister_device(mouse_dev);
317
318 out_input_free_device:
319   input_free_device(mouse_dev);
320
321 out_device_destroy:
```

```
322     device_destroy(dev_class, devnum);
323
324 out_class_destroy:
325     class_destroy(dev_class);
326
327 out_cdev_del:
328     cdev_del(&devs[0].cdev);
329
330 out_release_regions:
331     release_region(I8042_STATUS_REG + 1, 1);
332     release_region(I8042_DATA_REG + 1, 1);
333
334 out_unregister:
335     unregister_chrdev_region(devnum, VDEV_DEV_COUNT);
336
337 out:
338     return err;
339 }
340
341 static void __exit vdev_exit(void)
342 {
343     dev_t devnum = MKDEV(VDEV_MAJOR, VDEV_MINOR);
344
345     // Delete char device from system
346     cdev_del(&devs[0].cdev);
347
348     // Free IRQ
349     free_irq(I8042_KBD_IRQ, &devs[0]);
350
351     // Release I/O keybourd ports
352     release_region(I8042_STATUS_REG + 1, 1);
353     release_region(I8042_DATA_REG + 1, 1);
354
355     // Unregister char device
356     unregister_chrdev_region(devnum, VDEV_DEV_COUNT);
357
358     // Destroy struct class and device file
359     device_destroy(dev_class, devnum);
360     class_destroy(dev_class);
361
362     // Undregister input device
363     input_unregister_device(mouse_dev);
364
365     // Free input device
366     input_free_device(mouse_dev);
367
368     pr_notice("VDEV: Driver %s unloaded\n", MODULE_NAME);
369 }
```

```
370
371 module_init(vdev_init);
372 module_exit(vdev_exit);
```

Листинг 10. Листинг тестовой программы

```
1  #include <fcntl.h> // open
2  #include <stdio.h>
3  #include <stdlib.h> // EXIT_FAILURE
4  #include <string.h>
5  #include <unistd.h> // write, exit
6
7  #define DEVICE_PATH "/dev/VDEV"
8
9  void error(char* msg)
10 {
11   perror(msg);
12   exit(EXIT_FAILURE);
13 }
14
15 int main()
16 {
17   int fd;
18
19   fd = open(DEVICE_PATH, O_WRONLY);
20   if (fd < 0)
21     error("Device path not found");
22
23   // Write config to device
24   char* map = "0 edsfkl";
25   write(fd, map, strlen(map));
26
27   char* spd = "1 20";
28   write(fd, spd, strlen(spd));
29
30   close(fd);
31
32   return 0;
33 }
```