Страничка для ТЗ

Содержание

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 может просто выйти из строя. Так же не маловажным 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 ОС Linux, но достаточно часто они очень сложны для рядового пользователя или лишены некоторых ключевых функций.

Примеры:

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

2) MouseKeys - поддерживает только Ubuntu и требует наличие цифровой клавиатуры, и нет возможности изменить настройку по умолчинию.

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

1 Аналитический раздел

1.1 Постановка задачи

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.

1.2 Процесс обработки прерываний

Прерывание - это сообщение, информирующее систему о том, что одно из устройств выполнила 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.

1.2.1 Контролер 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

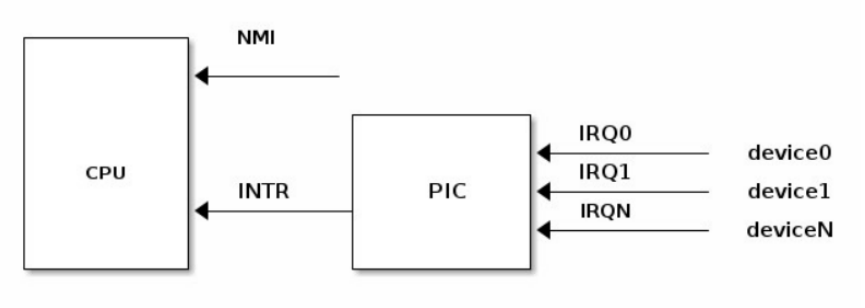

Рис. 1. Схема контролера прерываний

1.2.2 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> (Смотрите 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.

1.3 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 обработчике 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 softirqs. 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.

1.3.1 Вывод

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

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

1.4 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.

1.5 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].

1.5.1 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.

Нас интересует пространство ядра.

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

1.5.2 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

```
#inlcude <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 (*iopoll)(struct kiocb *kiocb, bool spin);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*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 *);
    unsigned long mmap_supported_flags;
    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 (*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);
```

```
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  }
```

1.5.3 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 освобождеия распредеения [7].

После присвоение идентификатора, символьное устройство должно быть инициализировано 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].

1.5.4 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.

1.6 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.

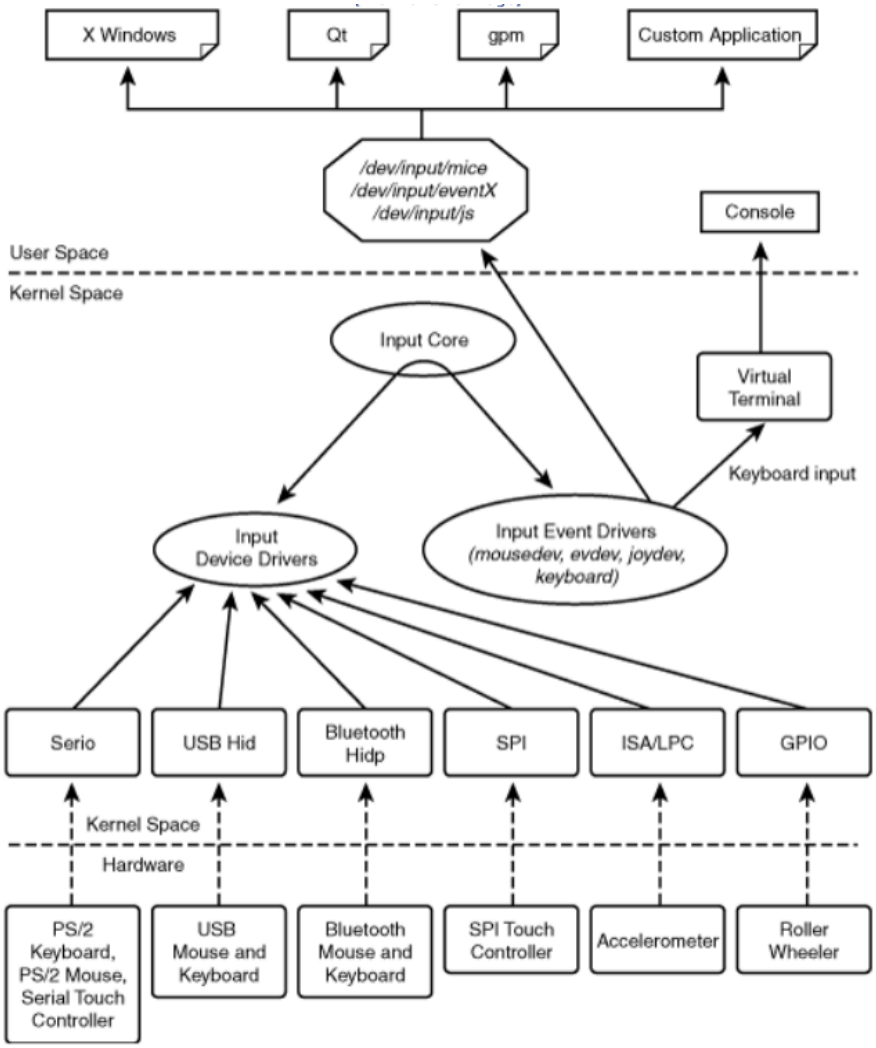

Рис. 2. Структура подсистемы ввода

1.6.1 Драйверы событий для мыши

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

```
1 struct input_event {
2     struct timeval time;     // Timestamp
3     \_\_u16 type;            // Event type
4     \_\_u16 code;                // Event code
5     \_\_s32 value;           // Event value
```

```
6 }
```

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.

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/uinpu* (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