Санкт-Петербургский Государственный Политехнический Университет

Институт информационных технологий и управления

Кафедра Компьютерных Систем и Программных Технологий

Отчет по лабораторной работе № 7

По дисциплине «Проектирование ОС и компонентов»

По теме «Разработка драйвера»

**Работу выполнила студентка группы № 13541/3**

Шаляпин Н.С. \_\_\_\_\_\_\_\_\_\_\_

**Работу принял преподаватель:**

Душутина Е.В. \_\_\_\_\_\_\_\_\_\_\_

.

Санкт-Петербург

2017

## Программа работы

Встраивание модулей в ядро. Разработка драйвера (на примере символьного устройства). Реализация различных (min двух) слоев, в том числе нижнего уровня с обработкой событий и обращением к портам.

Лабораторная работа выполнялась в ОС Ubuntu 16.04.

|  |
| --- |
| nikita@nikita-K53SM:~$ uname -a  Linux nikita@nikita-K53SM 4.4.0-31-generic #50-Ubuntu SMP Wed Jul 13 00:07:12 UTC 2016 x86\_64 x86\_64 x86\_64 GNU/Linux |

1. **Разработка драйвера символьного устройства**

Модуль ядра должен иметь как минимум 2 функции: функцию инициализации и функцию выхода. Первая вызывается во время загрузки модуля в пространство ядра, вторая при его выгрузке. Эти функции задаются с помощью: **module\_init** и **module\_exit**. Будет использоваться функция **printk()** для регистрации событий загрузки и выгрузки модуля. Для связи пространства ядра с пространством пользователя используются функции **put\_user(), get\_user().** Функция **put\_user()** перекидывает данные из пространства ядра в пользовательское, функция **get\_user()** совершает обратное действие. Пользователь может записывать сообщение с максимальной длиной 140 символов, 40 символов выделяется на сообщение "\nI already told you %d times HELLO!\n". Код chardev.c:

|  |
| --- |
| #include <linux/kernel.h>  #include <linux/module.h>  #include <linux/fs.h>  #include <asm/uaccess.h>  #include <linux/init.h>  #include <linux/slab.h>  #include <linux/errno.h>  #include <linux/types.h>  #define SUCCESS 0  #define DEVICE\_NAME "chardev" /\* Имя устройства, будет отображаться в /proc/devices \*/  #define BUF\_LEN 180 /\* Максимальная длина сообщения \*/  int device\_init(void);  void device\_exit(void);  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 \*);  module\_init(device\_init);  module\_exit(device\_exit);  /\* объявлены как статические \*/  static int Major; /\* Старший номер устройства - драйвера \*/  static int Device\_Open = 0; /\* Счетчик открытия устройства  \* Используется для предотвращения обращения  \* из нескольких процессов \*/  static char Message[BUF\_LEN]; /\* Текст сообщения \*/  static char \*Message\_Ptr;  static struct file\_operations fops = {  .read = device\_read,  .write = device\_write,  .open = device\_open,  .release = device\_release  };  // Ниже мы задаём информацию о модуле, которую можно будет увидеть с помощью Modinfo  MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Nikita Shalyapin");  /\* Загрузка модуля в ядро \*/  int device\_init(void)  {  /\* Регистрация устройства \*/  Major = register\_chrdev(0, DEVICE\_NAME, &fops);  if (Major < 0) {  printk(KERN\_ALERT "Registering char device failed with %d\n", Major);  return Major;  }  printk(KERN\_INFO "I was assigned major number %d. To talk to\n", Major);  printk(KERN\_INFO "the driver, create a dev file with\n");  printk(KERN\_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE\_NAME, Major);  printk(KERN\_INFO "Try various minor numbers. Try to cat and echo to\n");  printk(KERN\_INFO "the device file.\n");  printk(KERN\_INFO "Remove the device file and module when done.\n");  return SUCCESS;  }  /\* Выгрузка модуля из ядра \*/  void device\_exit(void)  {  /\* Освобождение старшего номера устройства \*/  unregister\_chrdev(Major, DEVICE\_NAME);  printk(KERN\_ALERT "Removing character device /dev/%s c %d 0'.\n", DEVICE\_NAME, Major);  return;  }  /\* Открытие файла устройства процессом "cat /dev/chardev" \*/  static int device\_open(struct inode \*inode, struct file \*file)  {  printk(KERN\_INFO "Try to open character device /dev/%s c %d 0'.\n", DEVICE\_NAME, Major);  if (Device\_Open)  return -EBUSY;  Device\_Open++; // счетчик открытия устройства  try\_module\_get(THIS\_MODULE);  return SUCCESS;  }  /\* Закрытие файла устройства процессом \*/  static int device\_release(struct inode \*inode, struct file \*file)  {  printk(KERN\_ALERT "Try to close character device /dev/%s c %d 0'.\n", DEVICE\_NAME, Major);  Device\_Open--; /\* Возможно обслуживание другого процесса \*/  /\*  \* Уменьшить счетчик обращений, иначе после успешного  \* открытия не сможем больше выгрузить модуль.  \*/  module\_put(THIS\_MODULE);  return SUCCESS;  }  /\* Открытие файла устройства процессом для чтения \*/  static ssize\_t device\_read(struct file \*filp, /\*указатель на структуру file\*/  char \*buffer, /\* буфер, куда надо положить данные \*/  size\_t length, /\* длина буфера \*/  loff\_t \* offset)  {  int bytes\_read = 0; /\* Количество байт, записанных в буфер \*/  printk("Try to read character device /dev/%s c %d 0'.\n", DEVICE\_NAME, Major);  if (\*Message\_Ptr == 0) /\* Если достигли конца сообщения, \*/  return 0; /\*вернуть ноль, как признак конца файла\*/  /\* Помещение данных в буфер\*/  while (length && \*Message\_Ptr) {  put\_user(\*(Message\_Ptr++), buffer++);  length--;  bytes\_read++;  }  printk("Read %d bytes, %d left.\n", bytes\_read, length);  /\* Возвращаем число байт, записанных в буфер \*/  return bytes\_read;  }  /\* Открытие файла устройства процессом для записи \*/  static ssize\_t device\_write(struct file \*filp, const char \*buffer, size\_t length, loff\_t \* offset)  {  static int counter = 0;  int i;  printk("Try to write character device /dev/%s (%d).\n", DEVICE\_NAME, length);  memset(Message, 0, BUF\_LEN);  for (i = 0; i < length && i < (BUF\_LEN - 40); i++)  get\_user(Message[i], buffer + i);  counter++;  sprintf(Message + strlen(Message), "\nI already told you %d times HELLO!\n", counter);  Message\_Ptr = Message;  return i;  } |

Для сборки модуля используется следующий **Makefile**:

|  |
| --- |
| obj-m += chardev.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 |

**Последовательность действий:**

1. Получение файла с расширением .ko;

2. Загрузка модуля chardev.ko в ядро системы Linux;

3. Создание файла символьного устройства;

4. Изменение прав доступа к файлу для поддержки возможности записи в файл;

5. Запись в файл устройства;

6. Чтение файла устройства;

7. Выгрузка модуля из ядра;

8. Удаление файла символьного устройства;

9. Очистка каталога от исполняемых файлов.

**Лог действий:**

|  |
| --- |
| **//1. Получение файла с расширением .ko**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ make  make -C /lib/modules/4.4.0-66-generic/build M=/home/nikita/labs/2semester/lab7/1 modules  make[1]: вход в каталог «/usr/src/linux-headers-4.4.0-66-generic»  CC [M] /home/nikita/labs/2semester/lab7/1/chardev.o  Building modules, stage 2.  MODPOST 1 modules  CC /home/nikita/labs/2semester/lab7/1/chardev.mod.o  LD [M] /home/nikita/labs/2semester/lab7/1/chardev.ko  make[1]: выход из каталога «/usr/src/linux-headers-4.4.0-66-generic»  **//2. Загрузка модуля chardev.ko в ядро системы Linux**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ sudo insmod chardev.ko  [sudo] пароль для nikita:  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ tail /var/log/syslog  May 3 15:03:31 nikita@nikita-K53SM kernel: [ 6361.615124] I was assigned major number 244. To talk to  May 3 15:03:31 nikita@nikita-K53SM kernel: [ 6361.615128] the driver, create a dev file with  May 3 15:03:31 nikita@nikita-K53SM kernel: [ 6361.615129] 'mknod /dev/chardev c 244 0'.  May 3 15:03:31 nikita@nikita-K53SM kernel: [ 6361.615130] Try various minor numbers. Try to cat and echo to  May 3 15:03:31 nikita@nikita-K53SM kernel: [ 6361.615131] the device file.  May 3 15:03:31 nikita@nikita-K53SM kernel: [ 6361.615132] Remove the device file and module when done.  **//3. Создание файла символьного устройства**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ sudo mknod /dev/chardev c 244 0  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ lsmod  Module Size Used by  chardev 16384 0  bnep 20480 2  ...  **//4. Изменение прав доступа к файлу для поддержки возможности записи в файл**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ ls -l /dev/chardev  crw-r--r-- 1 root root 249, 0 май 3 15:06 /dev/chardev  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ sudo chmod 666 /dev/chardev  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ ls -l /dev/chardev  crw-rw-rw- 1 root root 249, 0 май 3 15:06 /dev/chardev  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ cat /proc/devices  Character devices:  1 mem  ...  244 chardev  ...  **// Функционирование драйвера в консольном режиме использования**  **//5. Запись в файл устройства;**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ echo 'I say hello!' > /dev/chardev  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ tail /var/log/syslog  May 3 22:58:23 nikita@nikita-K53SM kernel: [ 877.393404] Try to write character device /dev/chardev (13).  May 3 22:58:23 nikita@nikita-K53SM kernel: [ 877.393408] Try to close character device /dev/chardev c 244 0'.  **//6. Чтение файла устройства**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ cat /dev/chardev  I say hello!  I already told you 1 times HELLO!  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ tail /var/log/syslog  May 3 22:58:24 nikita@nikita-K53SM kernel: [ 954.601486] Try to open character device /dev/chardev c 244 0'.  May 3 22:58:24 nikita@nikita-K53SM kernel: [ 954.601495] Try to read character device /dev/chardev c 244 0'.  May 3 22:58:24 nikita@nikita-K53SM kernel: [ 954.601498] Read 48 bytes, 131024 left.  May 3 22:58:24 nikita@nikita-K53SM kernel: [ 954.601506] Try to read character device /dev/chardev c 244 0'.  May 3 22:58:24 nikita@nikita-K53SM kernel: [ 954.601512] Try to close character device /dev/chardev c 244 0'.  **//7. Выгрузка модуля из ядра**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ sudo rmmod chardev.ko  [sudo] пароль для nikita:  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ tail /var/log/syslog  May 3 23:05:42 nikita@nikita-K53SM kernel: [ 1435.243711] Removing character device /dev/chardev c 244 0'.  **//8.Удаление файла устройства**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ sudo rm -i /dev/chardev  rm: удалить символьный специальный файл '/dev/chardev'? y  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ lsmod  Module Size Used by  bnep 20480 2  ...  **//9. Очистка каталога от исполняемых файлов**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ make clean  make -C /lib/modules/4.4.0-75-generic/build M=/home/nikita/labs/2semester/lab7/1 clean  make[1]: вход в каталог «/usr/src/linux-headers-4.4.0-75-generic»  CLEAN /home/nikita/labs/2semester/lab7/1/.tmp\_versions  CLEAN /home/nikita/labs/2semester/lab7/1/Module.symvers  make[1]: выход из каталога «/usr/src/linux-headers-4.4.0-75-generic»  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ ls -l  итого 12  -rw-rw-r-- 1 nikita nikita 5093 май 3 14:48 chardev.c  -rw-rw-r-- 1 nikita nikita 157 май 3 14:36 Makefile |

Был успешно загружен драйвер. Создан драйвер со старшим номером устройства 244. После этого был создан файл устройства с указанным номером. Теперь созданное устройство можно читать, а также в него можно записывать. При этом будет использоваться разработанный драйвер. Из результатов чтения и записи видно, что драйвер работает корректно.

Далее представлена программная реализация работы с драйвером (usedrv.c):

|  |
| --- |
| #include <fcntl.h>  #include <sys/stat.h>  #include <sys/types.h>  #include <stdlib.h>  #include <stdio.h>  #include <linux/unistd.h>  int main()  {  int fd;  size\_t cnt = 0;  size\_t cnt2 = 0;  char bufferIn[50] = "Hello, character device!";  char bufferOut[180];  printf("We work with character device driver\n");  fd = open("/dev/chardev", O\_RDWR);  if (fd == -1) {  printf("open failed\n");  return -1;  }  printf("Character driver open\n");    cnt = write(fd, bufferIn, sizeof(bufferIn));  printf("Character driver write %d bytes\n", cnt);  cnt = read(fd, bufferOut, sizeof(bufferOut));  printf("Character driver read %d bytes\n", cnt);  int i =0;  while ( i<cnt){  printf("%c",bufferOut[i]);  i++;  }  close(fd);  printf("\nCharacter driver close\n");  return 0;  } |

**Лог результата работы:**

|  |
| --- |
| **// Шаги 1-4 не отличаются от консольного варианта**  **// Результаты обращения к драйверу из программы (5, 6)**  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ gcc -o usedrv usedrv.c  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ ./usedrv  We work with character device driver  Character driver open  Character driver write 50 bytes  Character driver read 59 bytes  Hello, character device!  I already told you 1 times HELLO!  Character driver close  nikita@nikita-K53SM:~/labs/2semester/lab7/1$ tail /var/log/syslog  May 6 22:07:03 nikita@nikita-K53SM kernel: [ 584.054662] Try to close character device /dev/chardev c 244 0'.  May 6 22:07:45 nikita@nikita-K53SM kernel: [ 626.156380] Try to open character device /dev/chardev c 244 0'.  May 6 22:07:45 nikita@nikita-K53SM kernel: [ 626.156387] Try to write character device /dev/chardev (50).  May 6 22:07:45 nikita@nikita-K53SM kernel: [ 626.156402] Try to read character device /dev/chardev c 244 0'.  May 6 22:07:45 nikita@nikita-K53SM kernel: [ 626.156405] Read 59 bytes, 121 left.  May 6 22:07:45 nikita@nikita-K53SM kernel: [ 626.156418] Try to close character device /dev/chardev c 244 0'.  **// Шаги 7-9 не отличаются от консольного варианта** |

Программная реализация обращения к драйверу работает корректно: в символьное устройство была записана фраза и считана. Также символьное устройство дополнительно выводит сообщение, о том сколько раз в него была записана информация.

**2. Разработка модуля, обращающегося к порту клавиатуры**

Схема обработки аппаратных прерываний — это принципиально архитектурно зависимое действие, связанное с непосредственным взаимодействием с контроллером прерываний. Но схема в основных чертах остаётся неизменной, независимо от архитектуры. Вот как она выглядела, к примеру, в системе MS-DOS для процессоров x86 и контроллера прерываний (чип 8259) - на уровне ассемблера это нечто подобное последовательности действий:

* После возникновения аппаратного прерывания управление асинхронно получает функция, адрес которой записан в векторе прерывания.
* Обработку прерывания функция обработчика выполняет при запрещённых следующих прерываниях.
* После завершения обработки прерывания функция-обработчик восстанавливает контроллер прерываний, посылая сигнал о завершении прерывания. Это осуществляется отправкой команды EOI (End Of Interrupt — код 20h) в командный регистр микросхемы. Это однобайтовый регистр, который адресуется через порт ввода/вывода 20h.
* Функция-обработчик завершается, возвращая управление командой *iret*.

Показанная схема слишком архитектурно зависима (по взаимодействию с контроллером прерываний), даже с более современным чипом APIC контроллера процессора x86 схема взаимодействия в деталях будет выглядеть по-другому. Это недопустимо для многоплатформенной операционной системы, которой является Linux. Поэтому вводится логическая модель обработки прерываний, в которой аппаратно-зависимые элементы взаимодействия берёт на себя ядро, а обработка прерывания разделяется на две последовательные фазы:

* Регистрируется функция обработчика «верхней половины», который выполняется при запрещённых прерываниях локального процессора. Именно этой функции передаётся управление при возникновении аппаратного прерывания. Функция возвращает управление ядру системы традиционным *return*.
* Перед своим завершением функция-обработчик активирует последующее выполнение «нижней половины», которая и завершит позже начатую работу по обработке этого прерывания.
* В этой точке (после *return* из обработчика верхней половины) ядро завершает всё взаимодействие с аппаратурой контроллера прерываний, разрешает последующие прерывания, восстанавливает контроллер командой завершения обработки прерывания и возвращает управление из прерывания командой *iret*.
* Запланированная выше к выполнению функция нижней половины будет вызвана ядром в некоторый момент позже (но часто это может быть и непосредственно после завершения *return* из верхней половины), тогда, когда удобнее будет ядру системы. Принципиально важное отличие функции нижней половины состоит в том, что она выполняется уже при разрешённых прерываниях.

Исторически в Linux сменялось несколько разнообразных API реализации этой схемы (сами названия «верхняя половина» и «нижняя половина» - это дословно названия одной из старых схем, которая сейчас не присутствует в ядре). С появлением параллелизмов в ядре Linux, все новые схемы реализации обработчиков нижней половины (рассматриваются далее) построены на выполнении такого обработчика отдельным потоком ядра.

**2.1. Регистрация обработчика прерывания**

Функции и определения, реализующие интерфейс регистрации прерывания, объявлены в *<linux/interrupt.h>*. Первое, что мы должны всегда сделать — это зарегистрировать функцию обработчик прерываний:

typedef irqreturn\_t (\*irq\_handler\_t)( int, void\* );

int request\_irq ( unsigned int irq, irq\_handler\_t handler,

unsigned long flags,const char \*name, void \*dev );

extern void free\_irq( unsigned int irq, void \*dev );

где:

* *irq* - номер линии запрашиваемого прерывания;
* *handler* - указатель на функцию-обработчик;
* *flags* - битовая маска опций (описываемая далее), связанная с управлением прерыванием;
* *name* - символьная строка, используемая в */proc/interrupts*, для отображения владельца прерывания;
* *dev* - указатель на уникальный идентификатор устройства на линии IRQ, для не разделяемых прерываний (например, шины ISA) может указываться *NULL*. Данные по указателю *dev* требуются для удаления только специфицируемого устройства на разделяемой линии IRQ. Первоначально накладывалось единственное требование, чтобы этот указатель был уникальным, например, при размещении-освобождении N однотипных устройств вполне допустимым могла бы быть конструкция:

for( int i = 0; i < N; i++ ) request\_irq( irq, handler, 0, const char \*name,(void\*)i );

...

for( int i = 0; i < N; i++ ) free\_irq( irq, (void\*)i );

Но позже оказалось целесообразным и удобным использовать именно в качестве *\*dev* — указатель на специфическую для устройства структуру, которая и содержит все характерные данные экземпляра: поскольку для каждого экземпляра создаётся своя копия структуры, то указатели на них и будут уникальны, что и требовалось. На сегодня это общеупотребимая практика увязывать обработчик прерывания со структурами данных устройства. Флаги установки обработчика:

1. группа флагов установки обработчика по уровню (level-triggered) или фронту (edge-triggered):

*#define IRQF\_TRIGGER\_NONE 0x00000000*

*#define IRQF\_TRIGGER\_RISING 0x00000001*

*#define IRQF\_TRIGGER\_FALLING 0x00000002*

*#define IRQF\_TRIGGER\_HIGH 0x00000004*

*#define IRQF\_TRIGGER\_LOW 0x00000008*

*#define IRQF\_TRIGGER\_MASK ( IRQF\_TRIGGER\_HIGH | IRQF\_TRIGGER\_LOW |*

*IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING )*

*#define IRQF\_TRIGGER\_PROBE 0x00000010*

1. другие (не все, только основные, часто используемые) флаги:

*IRQF\_SHARED* — разрешить разделение (совместное использование) линии IRQ с другими устройствами (PCI шина и устройства). *IRQF\_PROBE\_SHARED* — устанавливается вызывающим, когда он предполагает возможные проблемы с совместным использованием. *IRQF\_TIMER* — флаг, маркирующий это прерывание как таймерное. *IRQF\_PERCPU* — прерывание, закреплённое монопольно за отдельным CPU. *IRQF\_NOBALANCING* — флаг, запрещающий вовлекать это прерывание в балансировку IRQ.

При успешной установке функция *request\_irq()* возвращает нуль. Возврат ненулевого значения указывает на то, что произошла ошибка, и указанный обработчик прерывания не был зарегистрирован. Наиболее часто встречающийся код ошибки — это значение -*EBUSY* (ошибки в ядре возвращаются отрицательными значениями), что указывает на то, что данная линия запроса на прерывание уже занята (или при текущем вызове, или при предыдущем вызове для этой линии не был указан флаг *IRQF\_SHARED*).

**2.2. Отображение прерываний в /proc**

Но, прежде чем дальше углубляться в организацию обработки прерывания, коротко остановимся на том, как мы можем наблюдать и контролировать то, что происходит с прерываниями. Всякий раз, когда аппаратное прерывание обрабатывается процессором, внутренний счётчик прерываний увеличивается, предоставляя возможность контроля за подсистемой прерываний; счётчики отображаются в */proc/interrupts* (последняя колонка это и есть имя обработчика, зарегистрированное параметром *name* в вызове *request\_irq()*). Те линии IRQ, для которых не установлены текущие обработчики прерываний, не отображаются в */proc/interrupts*.

Ещё одним источником (динамической) информации о произошедших (обработанных) прерываниях является файл */proc/stat.* Здесь строка, начинающаяся с *intr,* содержит суммарные по всем процессорам значения обработанных прерываний для всех последовательно линий IRQ.

**2.3. Обработчик прерываний, верхняя половина**

Прототип функции обработчика прерывания уже показывался выше:

*typedef irqreturn\_t (\*irq\_handler\_t)( int irq, void \*dev );*

где:

*irq* — линия IRQ;

*dev* — уникальный указатель экземпляра обработчика (именно тот, который передавался последним параметром *request\_irq()* при регистрации обработчика).

Это именно та функция, которая будет вызываться в первую очередь при каждом возникновении аппаратного прерывания. Но это вовсе не означает, что при возврате из этой функции работа по обработке текущего прерывания будет завершена (хотя и такой вариант вполне допустим). Из-за этой «неполноты» такой обработчик и получил название «верхняя половина» обработчика прерывания. Дальнейшие действия по обработке могут быть запланированы этим обработчиком на более позднее время, используя несколько различных механизмов, обобщённо называемых «нижняя половина».

Важно то, что код обработчика верхней половины выполняется при запрещённых последующих прерываниях по линии *irq* (этой же линии) для того локального процессора, на котором этот код выполняется. А после возврата из этой функции локальные прерывания будут вновь разрешены.

Возвращается значение (*<linux/irqreturn.h>*):

|  |
| --- |
| typedef int irqreturn\_t;  #define IRQ\_NONE (0)  #define IRQ\_HANDLED (1)  #define IRQ\_RETVAL(x) ((x) != 0) |

*IRQ\_HANDLED* — устройство прерывания распознано как обслуживаемое обработчиком, и прерывание успешно обработано.

*IRQ\_NONE* — устройство не является источником прерывания для данного обработчика, прерывание должно быть передано далее другим обработчикам, зарегистрированным на данной линии IRQ.

Типичная схема обработчика при этом будет выглядеть так:

static irqreturn\_t intr\_handler ( int irq, void \*dev ) {

if ( ! /\* проверка того, что обслуживаемое устройство запросило прерывание\*/ )

return IRQ\_NONE;

/\* код обслуживания устройства \*/

return IRQ\_HANDLED;

}

**2.4. Управление линиями прерывания**

Под управлением линиями прерываний будем понимать запрет-разрешение прерываний по одной или нескольким линиям *irq*. Раньше существовала возможность вообще запретить прерывания (на время). Но сейчас набор API для этих целей выглядит так: либо вы запрещаете прерывания по всем линиям *irq*, но локального процессора, либо на всех процессорах, но только для одной линии *irq*.

Макросы управления линиями прерываний определены в *<linux/irqflags.h>*. Управление запретом и разрешением прерываний на локальном процессоре:

* *local\_irq\_disable()* - запретить прерывания на локальном CPU;
* *local\_irq\_enable()* - разрешить прерывания на локальном CPU;
* *int irqs\_disabled()* - возвратить ненулевое значение, если запрещены прерывания на локальном CPU, в противном случае возвращается нуль.

Напротив, управление (запрет и разрешение) одной выбранной линией *irq*, но уже относительно всех процессоров в системе, делают макросы:

* *void disable\_irq(unsigned int irq)*
* *void disable\_irq\_nosync(unsigned int irq) -* обе эти функции запрещают прерывания с линии *irq* на контроллере (для всех CPU), причём, *disable\_irq()* не возвращается до тех пор, пока все обработчики прерываний, которые в данный момент выполняются, не закончат работу;
* *void enable\_irq(unsigned int irq)* - разрешаются прерывания с линии *irq* на контроллере (для всех CPU);
* *void synchronize\_irq(unsigned int irq)* - ожидает пока завершится обработчик прерывания от линии *irq* (если он выполняется), в принципе, хорошая идея — всегда вызывать эту функцию перед выгрузкой модуля использующего эту линию IRQ;

Вызовы функций *disable\_irq\*()* и *enable\_irq()* должны обязательно быть парными - каждому вызову функции запрещения линии должен соответствовать вызов функции разрешения. Только после последнего вызова функции *enable\_irq()* линия запроса на прерывание будет снова разрешена.

**2.5. Разработка модуля, обращающегося к порту клавиатуры**

Реализован модуль ядра, который обращается непосредственно к порту клавиатуры. Модуль ядра регистрирует события: прерывания на клавиатуре и нажатие клавиши Esc.

|  |
| --- |
| #include <linux/module.h>  #include <linux/init.h>  #include <linux/interrupt.h>  static int irq = 1, my\_dev\_id, irq\_counter = 0, esc\_counter = 0;  /\* Функция-обработчик прерывания\*/  static irqreturn\_t my\_interrupt( int irq, void \*dev\_id ) {  irq\_counter++;  printk( KERN\_INFO "In the ISR: counter = %d\n", irq\_counter );  static unsigned char scancode;  /\* Чтение скан кода нажатой клавиши \*/  scancode = inb (0x60);  /\*Логирование в сиcтемный журнал \*/  if (scancode == 0x01) /\*скан-код нажатия клавиши Esc\*/  {  esc\_counter++;  printk ("Press Esc!\n");  }  if (scancode == 0x81) /\*скан-код отпускания клавиши Esc\*/  printk ("Release Esc!\n");  return IRQ\_NONE; /\* we return IRQ\_NONE because we are just observing \*/  }  /\* Инициализация модуля и регистрация обработчика прерывания\*/  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;  }  /\* Удаление обработчика прерывания \*/  static void \_\_exit my\_exit( void ) {  synchronize\_irq( irq );  free\_irq( irq, &my\_dev\_id );  printk( KERN\_INFO "Successfully unloading, irq\_counter = %d, esc\_counter = %d\n", irq\_counter, esc\_counter );  }  module\_init( my\_init );  module\_exit( my\_exit );  MODULE\_AUTHOR( "Nikita Shalyapin" );  MODULE\_DESCRIPTION( "lab 7" );  MODULE\_LICENSE( "GPL v2" ); |

Этот код создает новый модуль ядра linux. Функция my\_interrupt – обработчик прерывания, который считает количество поступивших прерываний и печатает это число в буфер сообщений ядра, а также количество нажатий на кнопку Esc. 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. Далее с помощью системных макросов module\_init и module\_exit мы устанавливаем функции инициализации и завершения и указываем тип лицензии. Установлена линия IRQ клавиатуры (IRQ 1).

В примере обработчику прерывания передаются 2 параметра, но при желании можно передавать и 3 (\* regs – регистры, хранящиеся в стеке прерванного процесса):

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

Для чтения скан-кода нажатой клавиши используется функция *inb*, которая осуществляет чтение порта, переданного в качестве входного аргумента в функцию. Считывается 60 порт, так как этот порт содержит данные устройства (клавиатуры), порт 64 содержит информацию о статусе устройства.

Для сборки модуля используется следующий **Makefile**:

|  |
| --- |
| obj-m += mykeyboard.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 |

Проверка работоспособности модуля:

|  |
| --- |
| **//1. Получение файла с расширением .ko**  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ make  make -C /lib/modules/4.4.0-75-generic/build M=/home/nikita/labs/2semester/lab7/2 modules  make[1]: вход в каталог «/usr/src/linux-headers-4.4.0-75-generic»  CC [M] /home/nikita/labs/2semester/lab7/2/mykeyboard.o  /home/nikita/labs/2semester/lab7/2/mykeyboard.c: In function ‘my\_interrupt’:  /home/nikita/labs/2semester/lab7/2/mykeyboard.c:16:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]  static unsigned char scancode;  ^  Building modules, stage 2.  MODPOST 1 modules  CC /home/nikita/labs/2semester/lab7/2/mykeyboard.mod.o  LD [M] /home/nikita/labs/2semester/lab7/2/mykeyboard.ko  make[1]: выход из каталога «/usr/src/linux-headers-4.4.0-75-generic»  **//2. Загрузка модуля mykeyboard.ko в ядро системы Linux**  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ sudo insmod mykeyboard.ko  [sudo] пароль для nikita:  **//3. Просмотр буфера сообщений ядра и отображения обработчика my\_interrupt в /proc/interrupts**  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ dmesg | tail -n 20**  [ 789.303605] In the ISR: counter = 149  [ 789.406559] In the ISR: counter = 150  [ 789.411845] In the ISR: counter = 151  [ 790.191830] In the ISR: counter = 152  [ 790.247862] In the ISR: counter = 153  [ 792.037251] In the ISR: counter = 154  [ 792.116686] In the ISR: counter = 155  [ 793.670816] In the ISR: counter = 156  [ 793.670821] Press Esc!  [ 793.816843] In the ISR: counter = 157  [ 793.816847] Release Esc!  [ 794.336549] In the ISR: counter = 158  [ 794.339074] In the ISR: counter = 159  [ 794.475259] In the ISR: counter = 160  [ 794.479801] In the ISR: counter = 161  [ 794.817627] In the ISR: counter = 162  [ 794.819928] In the ISR: counter = 163  [ 794.933351] In the ISR: counter = 164  [ 794.938079] In the ISR: counter = 165  [ 795.545636] In the ISR: counter = 166  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ cat /proc/interrupts**  CPU0 CPU1 CPU2 CPU3  0: 21 0 0 0 IR-IO-APIC 2-edge timer  1: 0 34 464 16 IR-IO-APIC 1-edge i8042, **my\_interrupt**  ...  **//4. Выгрузка модуля из ядра**  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ sudo rmmod mykeyboard.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ dmesg | tail  [ 839.371670] In the ISR: counter = 260  [ 839.451274] In the ISR: counter = 261  [ 839.756040] In the ISR: counter = 262  [ 839.838407] In the ISR: counter = 263  [ 841.158366] In the ISR: counter = 264  [ 841.235219] In the ISR: counter = 265  [ 841.344693] In the ISR: counter = 266  [ 841.415813] In the ISR: counter = 267  [ 842.183442] In the ISR: counter = 268  [ 842.190401] Successfully unloading, irq\_counter = 268, esc\_counter = 3 |

Из результатов видно, что информация о прерываниях клавиатуры и о нажатии клавиши Esc успешно зафиксирована.

**2. 6. Тасклеты**

Предыдущая схема достаточно тяжеловесная, и в большинстве случаем её подменяют тасклеты — механизм на базе тех же *softirq* с двумя фиксированными индексами *HI\_SOFTIRQ* или *TASKLET\_SOFTIRQ*. Тасклеты это ни что иное, как частный случай реализации softirq. Тасклеты представляются (*<linux/interrupt.h>*) с помощью структуры:

|  |
| --- |
| struct tasklet\_struct {  struct tasklet\_struct \*next; /\* указатель на следующий тасклет в списке \*/  unsigned long state; /\* текущее состояние тасклета \*/  atomic\_t count; /\* счетчик ссылок \*/  void (\*func)(unsigned long); /\* функция-обработчик тасклета\*/  unsigned long data; /\* аргумент функции-обработчика тасклета \*/  }; |

Поле state может принимать только одно из значений: *0, TASKLET\_STATE\_SCHED, TASLET\_STATE\_RUN*. Значение *TASKLET\_STATE\_SCHED* указывает на то, что тасклет запланирован на выполнение, а значение *TASLET\_STATE\_RUN* — что тасклет выполняется.

|  |
| --- |
| enum {  TASKLET\_STATE\_SCHED, /\* Tasklet is scheduled for execution \*/  TASKLET\_STATE\_RUN /\* Tasklet is running (SMP only) \*/  }; |

Поле *count* используется как счетчик ссылок на тасклет. Если это значение не равно нулю, то тасклет запрещен и не может выполняться; если оно равно нулю, то тасклет разрешен и может выполняться в случае, когда он помечен как ожидающий выполнения.

Схематически код использования тасклета полностью повторяет структуру кода *softirq*. Инициализация тасклета при инициализации модуля:

|  |
| --- |
| struct xxx\_device\_struct { /\* Device-specific structure \*/  /\* ... \*/  struct tasklet\_struct tsklt;  /\* ... \*/  }  void \_\_init xxx\_init() {  struct xxx\_device\_struct \*dev\_struct;  /\* ... \*/  request\_irq( irq, xxx\_interrupt, 0, "xxx", NULL );  /\* Initialize tasklet \*/  tasklet\_init( &dev\_struct->tsklt, xxx\_analyze, dev );  } |

Для статического создания тасклета (и соответственно, обеспечения прямого доступа к нему) могут использоваться один из двух макросов:

|  |
| --- |
| DECLARE\_TASKLET( name, func, data )  DECLARE\_TASKLET\_DISABLED( name, func, data ); |

Оба макроса статически создают экземпляр структуры *struct tasklet\_struct* с указанным именем (*name*). Второй макрос создает тасклет, но устанавливает для него значение поля count, равное единице, и, соответственно, этот тасклет будет запрещен для исполнения. Макрос *DECLARE\_TASKLET( name, func, data )* эквивалентен:

|  |
| --- |
| struct tasklet\_struct namt = { NULL, 0, ATOMIC\_INIT(0), func, data ). |

Используется, что совершенно естественно, в точности тот же прототип функции обработчика тасклета, что и в случае отложенных прерываний (в моих примерах просто использована та же функция).

Для того чтобы запланировать тасклет на выполнение (обычно в обработчике прерывания), должна быть вызвана функция *tasklet\_schedule()*, которой в качестве аргумента передается указатель на соответствующий экземпляр структуры *struct tasklet\_struct*:

/\* The interrupt handler \*/

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

struct xxx\_device\_struct \*dev\_struct;

/\* ... \*/

/\* Mark tasklet as pending \*/

tasklet\_schedule( &dev\_struct->tsklt );

return IRQ\_HANDLED;

}

После того как тасклет запланирован на выполнение, он выполняется один раз в некоторый момент времени в ближайшем будущем. Для оптимизации тасклет всегда выполняется на том процессоре, который его запланировал на выполнение, что дает надежду на лучшее использование кэша процессора.

Если вместо стандартного тасклета нужно использовать тасклет высокого приоритета (*HI\_SOFTIRQ*), то вместо функции *tasklet\_schedule()* вызываем функцию планирования *tasklet\_hi\_schedule()*.

Уже запланированный тасклет может быть запрещен к исполнению (временно) с помощью вызова функции *tasklet\_disable()*. Если тасклет в данный момент уже начал выполнение, то функция не возвратит управление, пока тасклет не закончит своё выполнение. Как альтернативу можно использовать функцию *tasklet\_disable\_nosync()*, которая запрещает указанный тасклет, но возвращается сразу не ожидая, пока тасклет завершит выполнение (это обычно небезопасно, так как в данном случае нельзя гарантировать, что тасклет не закончил выполнение). Вызов функции *tasklet\_enable()* разрешает тасклет. Эта функция также должна быть вызвана для того, чтобы можно было выполнить тасклет, созданный с помощью макроса *DECLARE\_TASKLET\_DISABLED ()*. Из очереди тасклетов, ожидающих выполнения, тасклет может быть удален с помощью функции *tasklet\_kill()*. Так же как и в случае отложенных прерываний (на которых он построен), тасклет не может переходить в блокированное состояние.

**2.7. Демон ksoftirqd**

Обработка отложенных прерываний (softirq) и, соответственно, тасклетов осуществляется с помощью набора потоков пространства ядра (по одному потоку на каждый процессор). Потоки пространства ядра помогают обрабатывать отложенные прерывания, когда система перегружена большим количеством отложенных прерываний.

|  |
| --- |
| nikita@nikita-K53SM:~/lab2$ ps -ALf | grep 'ksoftirqd'  root 3 2 3 0 1 18:11 ? 00:00:00 [ksoftirqd/0]  root 13 2 13 0 1 18:11 ? 00:00:00 [ksoftirqd/1]  root 18 2 18 0 1 18:11 ? 00:00:00 [ksoftirqd/2]  root 23 2 23 0 1 18:11 ? 00:00:00 [ksoftirqd/3] |

Для каждого процессора существует свой поток. Каждый поток имеет имя в виде ksoftirqd/n, где n — номер процессора. Например, в двухпроцессорной системе будут запущены два потока с именами ksoftirqd/0 и ksoftirqd/1. To, что на каждом процессоре выполняется свой поток, гарантирует, что если в системе есть свободный процессор, то он всегда будет в состоянии выполнять отложенные прерывания. После того как потоки запущены, они выполняют замкнутый цикл.

**2.8. Очереди отложенных действий (workqueue)**

Очереди отложенных действий (*workqueue*) — это еще один, но совершенно другой, способ реализации отложенных операций. Очереди отложенных действий позволяют откладывать некоторые операции для последующего выполнения потоком пространства ядра (эти потоки ядра называют рабочими потоками - worker threads) — отложенные действия всегда выполняются в контексте процесса. Поэтому код, выполнение которого отложено с помощью постановки в очередь отложенных действий, получает все преимущества, которыми обладает код, выполняющийся в контексте процесса, главное из которых — это возможность переходить в блокированные состояния. Рабочие потоки, которые выполняются по умолчанию, называются *events/n*, где *n* — номер процессора, для 2-х процессоров это будут *events/0* и *events/1*:

|  |
| --- |
| **$ ps -ALf | head -n12**  ...  root 9 2 9 0 1 08:55 ? 00:00:00 [events/0]  root 10 2 10 0 1 08:55 ? 00:00:00 [events/1]  ... |

Когда какие-либо действия ставятся в очередь, поток ядра возвращается к выполнению и выполняет эти действия. Когда в очереди не остается работы, которую нужно выполнять, поток снова возвращается в состояние ожидания. Каждое действие представлено с помощью *struct work\_struct*:

|  |
| --- |
| typedef void (\*work\_func\_t)( struct work\_struct \*work );  struct work\_struct {  atomic\_long\_t data; /\* аргумент функции-обработчика \*/  struct list\_head entry; /\* связанный список всех действий \*/  work\_func\_t func; /\* функция-обработчик \*/  ...  }; |

Для создания статической структуры действия на этапе компиляции необходимо использовать макрос:

|  |
| --- |
| DECLARE\_WORK( name, void (\*func) (void \*), void \*data ); |

Это выражение создает *struct work\_struct* с именем *name*, с функцией-обработчиком *func()* и аргументом функции-обработчика *data*. Динамически отложенное действие создается с помощью указателя на ранее созданную структуру, используя следующий макрос:

|  |
| --- |
| INIT\_WORK( struct work\_struct \*work, void (\*func)(void \*), void \*data ); |

Функция-обработчика имеет тот же прототип, что и для отложенных прерываний и тасклетов, поэтому в примерах будет использоваться та же функция (*xxx\_analyze()*).

Для реализации нижней половины обработчика IRQ на технике *workqueue*, выполним последовательность действий примерно следующего содержания.

При инициализации модуля создаём отложенное действие:

|  |
| --- |
| #include <linux/workqueue.h>  struct work\_struct \*hardwork;  void \_\_init xxx\_init() {  /\* ... \*/  request\_irq( irq, xxx\_interrupt, 0, "xxx", NULL );  hardwork = kmalloc( sizeof(struct work\_struct), GFP\_KERNEL );  /\* Init the work structure \*/  INIT\_WORK( hardwork, xxx\_analyze, data );  } |

Или то же самое может быть выполнено статически:

|  |
| --- |
| #include <linux/workqueue.h>  DECLARE\_WORK( hardwork, xxx\_analyze, data );  void \_\_init xxx\_init() {  /\* ... \*/  request\_irq( irq, xxx\_interrupt, 0, "xxx", NULL );  } |

Самая интересная работа начинается, когда нужно запланировать отложенное действие; при использовании для этого рабочего потока ядра по умолчанию (*events/n*) это делается функциями:

* *schedule\_work( struct work\_struct \*work );* - действие планируется на выполнение немедленно и будет выполнено, как только рабочий поток *events*, работающий на данном процессоре, перейдет в состояние выполнения.
* *schedule\_delayed\_work( struct delayed\_work \*work, unsigned long delay );* - в этом случае запланированное действие не будет выполнено, пока не пройдет хотя бы заданное в параметре *delay* количество импульсов системного таймера.

В обработчике прерывания это выглядит так:

|  |
| --- |
| static irqreturn\_t xxx\_interrupt( int irq, void \*dev\_id ) {  /\* ... \*/  schedule\_work( hardwork );  /\* или schedule\_work( &hardwork ); - для статической инициализации \*/  return IRQ\_HANDLED;  } |

Очень часто бывает необходимо ждать, пока очередь отложенных действий очистится (отложенные действия завершатся), это обеспечивает функция:

|  |
| --- |
| void flush\_scheduled\_work( void ); |

Для отмены незавершённых отложенных действий с задержками используется функция:

|  |
| --- |
| int cancel\_delayed\_work( struct work\_struct \*work ); |

Но мы не обязательно должны рассчитывать на общие очереди (потоки ядра *events*) для выполнения отложенных действий — мы можем создать под эти цели собственные очереди (вместе с обслуживающим потоком). Создание обеспечивается макросами вида:

|  |
| --- |
| struct workqueue\_struct \*create\_workqueue( const char \*name );  struct workqueue\_struct \*create\_singlethread\_workqueue( const char \*name ); |

Планирование на выполнение в этом случае осуществляют функции:

|  |
| --- |
| int queue\_work( struct workqueue\_struct \*wq, struct work\_struct \*work );  int queue\_delayed\_work( struct workqueue\_struct \*wq,struct wesrk\_struct \*work, unsigned long delay); |

Они аналогичны рассмотренным выше *schedule\_\*()*, но работают с созданной очередью, указанной 1-м параметром. С вновь созданными потоками предыдущий пример может выглядеть так:

|  |
| --- |
| struct workqueue\_struct \*wq;  /\* Driver Initialization \*/  static int \_\_init xxx\_init( void ) {  /\* ... \*/  request\_irq( irq, xxx\_interrupt, 0, "xxx", NULL );  hardwork = kmalloc( sizeof(struct work\_struct), GFP\_KERNEL );  /\* Init the work structure \*/  INIT\_WORK( hardwork, xxx\_analyze, data );  wq = create\_singlethread\_workqueue( "xxxdrv" );  return 0;  }  static irqreturn\_t xxx\_interrupt( int irq, void \*dev\_id ) {  /\* ... \*/  queue\_work( wq, hardwork );  return IRQ\_HANDLED;  } |

Аналогично тому, как и для очереди по умолчанию, ожидание завершения действий в заданной очереди может быть выполнено с помощью функции:

|  |
| --- |
| void flush\_workqueue( struct workqueue\_struct \*wq ); |

**2.9. Сравнение и примеры**

Начнём со сравнений. Оставив в стороне рассмотрение *softirq*, как механизм тяжёлый, и уже достаточно обсуждённый, в том смысле, что его использование оправдано при требовании масштабирования высокоскоростных процессов на большое число обслуживающих процессоров в SMP. Две другие рассмотренные схемы — это тасклеты и очереди отложенных действий. Они представляют две различные схемы реализации отложенных работ в современном Linux, которые переносят работы из верхних половин в нижние половины драйверов. В тасклетах реализуется механизм с низкой латентностью, который является простым и ясным, а очереди работ имеют более гибкий и развитый API, который позволяет обслуживать несколько отложенных действий в порядке очередей. В каждой схеме откладывание (планирование) последующей работы выполняется из контекста прерывания, но только тасклеты выполняют запуск автоматически в стиле «работа до полного завершения», тогда как очереди отложенных действий разрешают функциям-обработчикам переходить в блокированные состояния. В этом состоит главное принципиальное отличие: рабочая функция тасклета не может блокироваться. Ниже показан пример для тасклетов:

|  |
| --- |
| **mod\_tasklet.c :**  #include <linux/module.h>  #include <linux/jiffies.h>  #include <linux/interrupt.h>  #include <linux/timex.h>  MODULE\_LICENSE("GPL");  cycles\_t cycles1, cycles2;  static u32 j1, j2;  char tasklet\_data[] = "tasklet\_function was called";  /\* Bottom Half Function \*/  void tasklet\_function( unsigned long data ) {  j2 = jiffies;  // В ядре Linux время измеряется с помощью глобальной переменной с именем // jiffies, которая определяет количество временных тиков (тактов)  cycles2 = get\_cycles();  printk( "%010lld [%05d] : %s\n", (long long unsigned)cycles2, j2, (char\*)data );  return;  }  DECLARE\_TASKLET( my\_tasklet, tasklet\_function, (unsigned long)&tasklet\_data );  int init\_module( void ) {  j1 = jiffies;  cycles1 = get\_cycles();  printk( "%010lld [%05d] : tasklet\_scheduled\n", (long long unsigned)cycles1, j1 );  /\* Schedule the Bottom Half \*/  tasklet\_schedule( &my\_tasklet );  return 0;  }  void cleanup\_module( void ) {  /\* Stop the tasklet before we exit \*/  tasklet\_kill( &my\_tasklet );  return;  } |

Вот как выглядит его исполнение:

|  |
| --- |
| **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ make**  make -C /lib/modules/4.4.0-75-generic/build M=/home/nikita/labs/2semester/lab7/2 modules  make[1]: вход в каталог «/usr/src/linux-headers-4.4.0-75-generic»  CC [M] /home/nikita/labs/2semester/lab7/2/mod\_tasklet.o  CC [M] /home/nikita/labs/2semester/lab7/2/mod\_workqueue.o  Building modules, stage 2.  MODPOST 3 modules  CC /home/nikita/labs/2semester/lab7/2/mod\_tasklet.mod.o  LD [M] /home/nikita/labs/2semester/lab7/2/mod\_tasklet.ko  CC /home/nikita/labs/2semester/lab7/2/mod\_workqueue.mod.o  LD [M] /home/nikita/labs/2semester/lab7/2/mod\_workqueue.ko  make[1]: выход из каталога «/usr/src/linux-headers-4.4.0-75-generic»  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ sudo insmod mod\_tasklet.ko**  [sudo] пароль для nikita:  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ dmesg | tail -n100 | grep " : "**  [ 4991.782392] 10971008153150 [1172957] : tasklet\_scheduled  [ 4991.782513] 10971008421646 [1172957] : tasklet\_function was called  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ sudo rmmod mod\_tasklet**  [ 4956.924387] 2188683488508 [1164268] : tasklet\_scheduled  [ 4956.924461] 2188683651047 [1164268] : tasklet\_function was called  **nikita@nikita-K53SM:~/lab2$ sudo rmmod mod\_tasklet** |

По временным меткам видно, что выполнение функции тасклета происходит позже планирования тасклета на выполнение, но латентность очень низкая (системный счётчик *jiffies* не успевает изменить значение, всё происходит в пределах одного системного тика), отсрочка выполнения составляет порядка 270000 процессорных тактов. В следующем примере мы проделаем практически то же самое (близкие эксперименты для возможностей сравнения), но относительно очередей отложенных действий:

|  |
| --- |
| **mod\_workqueue.c :**  #include <linux/module.h>  #include <linux/jiffies.h>  #include <linux/interrupt.h>  #include <linux/timex.h>  #include<linux/slab.h>  MODULE\_LICENSE("GPL");  static struct workqueue\_struct \*my\_wq;  typedef struct {  struct work\_struct my\_work;  int id;  u32 j;  cycles\_t cycles;  } my\_work\_t;  /\* Bottom Half Function \*/  static void my\_wq\_function( struct work\_struct \*work ) {  u32 j = jiffies;  cycles\_t cycles = get\_cycles();  my\_work\_t \*wrk = (my\_work\_t\*) work;  printk( "#%d : %010lld [%05d] => %010lld [%05d]\n",  wrk->id,  (long long unsigned)wrk->cycles, wrk->j,  (long long unsigned)cycles, j  );  kfree( (void \*)wrk );  return;  }  int init\_module( void ) {  my\_work\_t \*work1, \*work2;  int ret;  my\_wq = create\_workqueue( "my\_queue" );  if( my\_wq ) {  /\* Queue some work (item 1) \*/  work1 = (my\_work\_t\*)kmalloc( sizeof(my\_work\_t), GFP\_KERNEL );  if( work1 ) {  INIT\_WORK( (struct work\_struct \*)work1, my\_wq\_function );  work1->id = 1;  work1->j = jiffies;  work1->cycles = get\_cycles();  ret = queue\_work( my\_wq, (struct work\_struct \*)work1 );  }  /\* Queue some additional work (item 2) \*/  work2 = (my\_work\_t\*)kmalloc( sizeof(my\_work\_t), GFP\_KERNEL );  if( work2 ) {  INIT\_WORK( (struct work\_struct \*)work2, my\_wq\_function );  work2->id = 2;work2->j = jiffies;  work2->cycles = get\_cycles();  ret = queue\_work( my\_wq, (struct work\_struct \*)work2 );  }  }  return 0;  }  void cleanup\_module( void ) {  flush\_workqueue( my\_wq );  destroy\_workqueue( my\_wq );  return;  } |

Результаты выполнения:

|  |
| --- |
| **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ make**  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ sudo insmod mod\_workqueue.ko**  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ lsmod | head -n3**  Module Size Used by  **mod\_workqueue**  16384 0  bnep 20480 2  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ ps -ef | grep my\_**  root 20710 2 0 13:42 ? 00:00:00 [my\_queue]  nikita 20801 4759 0 13:42 pts/0 00:00:00 grep --color=auto my\_  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ dmesg | grep "=>"**  [ 0.000000] e820: update [mem 0x00000000-0x00000fff] usable ==> reserved  [ 0.000000] e820: update [mem 0x9cfff000-0xffffffff] usable ==> reserved  [ 5239.877686] #1 : 11515558713435 [1234983] => 11515558717398 [1234983]  [ 5239.877690] #2 : 11515558727807 [1234983] => 11515558729806 [1234983]  **nikita@nikita-K53SM:~/labs/2semester/lab7/2$ sudo rmmod mod\_workqueue** |

Из результатов видно, как появился новый обрабатывающий поток ядра, с заданным нами именем. На этот раз мы помещаем в очередь отложенных действий два экземпляра работы, здесь латентность реакции существенно больше случая тасклетов.

**2.10. Разработка модуля, обращающегося к порту клавиатуры. Использование очереди отложенных действий (workqueue)**

Реализован модуль ядра, который обращается непосредственно к порту клавиатуры. Модуль ядра регистрирует события: прерывания на клавиатуре и нажатие клавиши Esc.

|  |
| --- |
| **mod\_workqueue.c**  #include <linux/module.h>  #include <linux/init.h>  #include <linux/interrupt.h>  #include <linux/jiffies.h>  #include <linux/timex.h>  #include<linux/slab.h>  static int irq = 1, my\_dev\_id, irq\_counter = 0, esc\_counter = 0;  static struct workqueue\_struct \*my\_wq;  typedef struct {  struct work\_struct my\_work;  int id;  u32 j;  cycles\_t cycles;  int irq\_cnt;  } my\_work\_t;  /\* Bottom Half Function \*/  static void my\_wq\_function( struct work\_struct \*work ) {  u32 j = jiffies;  cycles\_t cycles = get\_cycles();  my\_work\_t \*wrk = (my\_work\_t\*) work;  printk( "#%d : %010lld [%05d] => %010lld [%05d]\n",  wrk->id,  (long long unsigned)wrk->cycles, wrk->j,  (long long unsigned)cycles, j  );  printk( KERN\_INFO "In the ISR: counter = %d\n", wrk->irq\_cnt );  kfree( (void \*)wrk );  return;  }  /\* Функция-обработчик прерывания\*/  static irqreturn\_t my\_interrupt( int irq, void \*dev\_id ) {  my\_work\_t \*work1;  irq\_counter++;  int ret;  static unsigned char scancode;  /\* Чтение скан кода нажатой клавиши \*/  scancode = inb (0x60);  /\*Логирование в сиcтемный журнал \*/  if (scancode == 0x01) /\*скан-код нажатия клавиши Esc\*/  {  esc\_counter++;  printk ("Press Esc!\n");  }  if (scancode == 0x81) /\*скан-код отпускания клавиши Esc\*/  printk ("Release Esc!\n");  if( my\_wq ) {  /\* Queue some work (item 1) \*/  work1 = (my\_work\_t\*)kmalloc( sizeof(my\_work\_t), GFP\_KERNEL );  if( work1 ) {  INIT\_WORK( (struct work\_struct \*)work1, my\_wq\_function );  work1->id = 1;  work1->j = jiffies;  work1->cycles = get\_cycles();  work1->irq\_cnt = irq\_counter;  ret = queue\_work( my\_wq, (struct work\_struct \*)work1 );  }  }  return IRQ\_NONE; /\* we return IRQ\_NONE because we are just observing \*/  }  /\* Инициализация модуля и регистрация обработчика прерывания\*/  static int \_\_init my\_init( void ) {  my\_wq = create\_workqueue( "my\_queue" );  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;  }  /\* Удаление обработчика прерывания \*/  static void \_\_exit my\_exit( void ) {  flush\_workqueue( my\_wq );  destroy\_workqueue( my\_wq );  synchronize\_irq( irq );  free\_irq( irq, &my\_dev\_id );  printk( KERN\_INFO "Successfully unloading, irq\_counter = %d, esc\_counter = %d\n", irq\_counter, esc\_counter );  }  module\_init( my\_init );  module\_exit( my\_exit );  MODULE\_AUTHOR( "Nikita Shalyapin" );  MODULE\_DESCRIPTION( "lab 7" );  MODULE\_LICENSE( "GPL v2" ); |

Этот код создает новый модуль ядра linux. Функция my\_interrupt – обработчик прерывания, который считает количество поступивших прерываний клавиатуры, а также количество нажатий на кнопку Esc, а также ставит в очередь работу, которая запишет в буфер ядра сообщение о количестве прерываний, а также системный счётчик (*jiffies)* и количество процессорных тактов при постановке в очередь и выполнении работы из очереди. Функция my\_init – функция инициализации, которая вызывается при загрузке модуля в ядро, а также создает очередь отложенных действий. 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 ждем завершения отложенных действий (flush\_workqueue), удаляем очередь отложенных действий (destroy\_workqueue), ожидаем освобождения обработчика (synchronize\_irq), снимаем свой обработчик (free\_irq) и печатаем соответствующее сообщение. Далее с помощью системных макросов module\_init и module\_exit мы устанавливаем функции инициализации и завершения и указываем тип лицензии. Установлена линия IRQ клавиатуры (IRQ 1).

Проверка работоспособности модуля:

|  |
| --- |
| **//1. Получение файла с расширением .ko**  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ make  make -C /lib/modules/4.4.0-75-generic/build M=/home/nikita/labs/2semester/lab7/2 modules  make[1]: вход в каталог «/usr/src/linux-headers-4.4.0-75-generic»  CC [M] /home/nikita/labs/2semester/lab7/2/mykeyboard\_workqueue.o  /home/nikita/labs/2semester/lab7/2/mykeyboard\_workqueue.c: In function ‘my\_interrupt’:  /home/nikita/labs/2semester/lab7/2/mykeyboard\_workqueue.c:37:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]  int ret;  ^  Building modules, stage 2.  MODPOST 4 modules  CC /home/nikita/labs/2semester/lab7/2/mykeyboard\_workqueue.mod.o  LD [M] /home/nikita/labs/2semester/lab7/2/mykeyboard\_workqueue.ko  make[1]: выход из каталога «/usr/src/linux-headers-4.4.0-75-generic»  **//2. Загрузка модуля в ядро системы Linux**  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ sudo insmod mykeyboard\_workqueue.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ **lsmod | head -n3**  Module Size Used by  **mykeyboard\_workqueue** 16384 0  bnep 20480 2  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ **ps -ef | grep my\_**  root 26215 2 0 14:05 ? 00:00:00 [**my\_queue**]  nikita 26250 4759 0 14:05 pts/0 00:00:00 grep --color=auto my\_  **//3. Просмотр буфера сообщений ядра**  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ dmesg | tail -n 20  [ 6716.120566] #1 : 14755801100120 [1604049] => 14755801118912 [1604049]  [ 6716.120568] In the ISR: counter = 136  [ 6716.186817] #1 : 14755946497776 [1604065] => 14755946535304 [1604066]  [ 6716.186819] In the ISR: counter = 137  [ 6716.356015] #1 : 14756317895508 [1604108] => 14756317913500 [1604108]  [ 6716.356018] In the ISR: counter = 138  [ 6716.419285] #1 : 14756456768680 [1604124] => 14756456785976 [1604124]  [ 6716.419288] In the ISR: counter = 139  [ 6717.798348] Press Esc!  [ 6717.798361] #1 : 14759483738956 [1604468] => 14759483757740 [1604468]  [ 6717.798362] In the ISR: counter = 140  [ 6717.908493] Release Esc!  [ 6717.908506] #1 : 14759725499368 [1604496] => 14759725518176 [1604496]  [ 6717.908507] In the ISR: counter = 141  [ 6718.335420] #1 : 14760662540876 [1604603] => 14760662559300 [1604603]  [ 6718.335423] In the ISR: counter = 142  [ 6718.421878] #1 : 14760852310116 [1604624] => 14760852327112 [1604624]  [ 6718.421880] In the ISR: counter = 143  [ 6719.861370] #1 : 14764011886512 [1604984] => 14764011903652 [1604984]  [ 6719.861373] In the ISR: counter = 144  **//4. Выгрузка модуля из ядра**  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ sudo rmmod mykeyboard\_workqueue.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ dmesg | tail  [ 6727.327536] In the ISR: counter = 166  [ 6727.330017] #1 : 14780405005676 [1606851] => 14780405024296 [1606851]  [ 6727.330019] In the ISR: counter = 167  [ 6727.406830] #1 : 14780573536540 [1606871] => 14780573619508 [1606871]  [ 6727.406833] In the ISR: counter = 168  [ 6727.413363] #1 : 14780587943676 [1606872] => 14780587961300 [1606872]  [ 6727.413365] In the ISR: counter = 169  [ 6731.506705] #1 : 14789557019006 [1607894] => 14789572541150 [1607896]  [ 6731.506716] In the ISR: counter = 170  [ 6731.506763] Successfully unloading, irq\_counter = 170, esc\_counter = 2  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ lsmod | head -n3  Module Size Used by  bnep 20480 2  ipt\_MASQUERADE 16384 1  nikita@nikita-K53SM:~/labs/2semester/lab7/2$ ps -ef | grep my\_  nikita 26550 4759 0 14:07 pts/0 00:00:00 grep --color=auto my\_ |

Из результатов видно, что информация о прерываниях клавиатуры и о нажатии клавиши Esc успешно зафиксирована. Для вывода сообщений о количестве прерываний клавиатуры была использована очередь отложенных действий (нижняя половина).

**3. Модуль ядра usb-мыши**

Реализуем модуль ядра, который будет обрабатывать прерывания самостоятельно, не используя информацию, полученную от драйвера, подключенного в систему по умолчанию. В данном примере в качестве устройства используется usb мышь, регистрируемые события ­ – нажатие на клавиши мыши и запись в системный журнал информации о событии (нажатии на правую кнопку, нажатии на левую кнопку, другое действие). Также будет подсчитываться количество нажатий на правую и левую кнопку.

Одна из главных концепций USB заключается в том, что в USB-системе может быть только один мастер. Им является host-компьютер. USB - устройства всегда отвечают на запросы host-компьютера - они никогда не могут посылать информацию самостоятельно.

Хост всегда является мастером, а обмен данными должен осуществляться в обоих направлениях:

\* OUT - отсылая пакет с флагом OUT, хост отсылает данные устройству

\* IN - отсылая пакет с флагом IN, хост отправляет запрос на прием данных из устройства. Чтобы принять данные из устройства, хост отсылает пакет с флагом IN.

**Endpoint - источник/приемник данных**

Спецификация USB определеят endpoint (EP), как источник или приемник данных. Устройство может иметь до 32 EP: 16 на прием и 16 на передачу.

Обращение к тому или иному endpoint'у происходит по его адресу.

**Регистрация/выгрузка драйвера.**

Регистрация USB-драйвера подразумевает:

1. заполнение структуры usb\_driver

2. регистрацию структуры в системе

Структура usb\_driver описана в include/linux/usb.h Рассмотрим наиболее важные поля этой структуры.

|  |
| --- |
| struct usb\_driver {  // ...  const char \*name;  int (\*probe) (struct usb\_interface \*intf,  const struct usb\_device\_id \*id);  void (\*disconnect) (struct usb\_interface \*intf);  const struct usb\_device\_id \*id\_table;  struct device\_driver driver;  // ...  }; |

Поле  **name** - это имя драйвера.

**id\_table** - это массив структур usb\_device\_id. Этот список предназначен для определения соответствия подключаемого устройства определенным параметрам. Только те устройства, которые соответствуют перечисленным параметрам, могут быть подключены к драйверу. Если массив пуст, система будет пытаться подключить каждое устройство к драйверу.

Поле **driver** говорит о том, что **usb\_driver** унаследован от **device\_driver**.

В самом простом случае каждый элемент id\_table[i] содержит пару идентификаторов:

\* идентификатор производителя (Vendor ID)

\* идентификатор устройства (Device ID).

**probe** и **disconnect** - это callback-функции, вызываемые системой при подключении и отключении USB-устройства. probe будет вызван для каждого устройства, если список id\_table пуст, или только для тех устройств, которые соответствуют параметрам, перечисленным в списке.

**Регистрация устройства**

Один зарегистрированный драйвер может "подключать" несколько устройств. Для подключения устройства к драйверу система вызывает функцию драйвера **probe**, которой передает 2 параметра:

|  |
| --- |
| static int my\_probe(struct usb\_interface \*interface, const struct usb\_device\_id \*id) |

**interface** - это интерфейс USB-устройства. Обычно USB-драйвер взаимодействует не с устройством напрямую, а с его интерфейсом. **id** - содержит информацию об устройстве. Если функция возвращает 0, то устройство успешно зарегистрировано, иначе - система попытается "привязать" устройство к какому-нибудь другому драйверу.

Для отключения устройства от драйвера система вызывает функцию **disconnect**, которой передается один параметр - интерфейс:

|  |
| --- |
| static void my\_disconnect(struct usb\_interface \*interface) |

В общем случае, в функции **probe** для каждого подключаемого устройства выделяется структура в памяти, заполняется, затем регистрируется, например, символьное устройство, и проводится регистрация устройства в sysfs.

**Использование USB Major**

Для регистрации символьного устройства необходимо получить число major - либо статически, либо динамически (вызвать register\_chrdev с параметром major = 0). Когда символьное устройство зарегистрировано, необходимо создать файл в директории /dev. Для этого можно воспользоваться командой mknod (из user-space).

В программном интерфейсе USB для этих целей есть функция **usb\_register\_dev**.

|  |
| --- |
| extern int usb\_register\_dev(struct usb\_interface \*intf, struct usb\_class\_driver \*class\_driver);  extern void usb\_deregister\_dev(struct usb\_interface \*intf, struct usb\_class\_driver \*class\_driver); |

**usb\_register\_dev** принимает на вход interface и class\_driver. Структура **usb\_class\_driver** выглядит следующим образом:

|  |
| --- |
| struct usb\_class\_driver {  char \*name;  struct file\_operations \*fops;  int minor\_base;  }; |

**name** - это имя устройства. Директория с этим именем появится в sysfs. **fops** - файловые операции символьного устройства. **minor\_base** - базовый minor номер.

Функция **usb\_register\_dev** выполняет следующие действия:

\* регистрирует символьное устройство с major номером 180 (см include/linux/major.h) и резервирует диапазон из 16 minor-номеров. Поэтому minor\_base должен иметь младший полубайт = 0.

\* в зарезервированном диапазоне minor номеров выделяет один номер для данного устройства. Этот номер записывает в interface->minor.

\* создает все необходимые файлы в sysfs: после этого udev создает файлы в /dev/

Вызов **usb\_deregister\_dev** выполняет обратные процедуры, поэтому должен вызываться в функции disconnect.

**Установка функции-обработчика прерывания**

|  |
| --- |
| usb\_fill\_int\_urb(mouse->irq, dev, pipe, mouse→data, (maxp > 8 ? 8 : maxp), usb\_mouse\_irq, mouse, endpoint->bInterval); |

Параметры:

* mouse->irq – указатель на структуру urb (USB Request Block) – структура для передачи и приема данных с USB устройством;
* dev – указатель на usb\_device
* pipe – конкретный endpoint, к которому должен быть отправлен urb;
* mouse->data – буфер, с которого поступают входящие данные или из которого забираются исходящие данные;
* (maxp > 8 ? 8 : maxp) – длина буфера (предыдущее поле);
* usb\_mouse\_irq – указатель на функцию-обработчик прерывания;
* mouse – указатель на дополнительный объект;
* endpoint->bInterval – заданный интервал.

Код драйвера:

|  |
| --- |
| #include <linux/kernel.h>  #include <linux/slab.h>  #include <linux/module.h>  #include <linux/init.h>  #include <linux/usb/input.h>  #include <linux/hid.h>  #include <linux/fs.h>  #include <asm/segment.h>  #include <asm/uaccess.h>  #include <linux/buffer\_head.h>  #include <linux/device.h>  #include <linux/cdev.h>  #define DRIVER\_AUTHOR "Fomina Nikita"  #define DRIVER\_LICENSE "GPL"  MODULE\_AUTHOR(DRIVER\_AUTHOR);  MODULE\_LICENSE(DRIVER\_LICENSE);  static int btn\_right\_counter = 0;  static int btn\_left\_counter = 0;  struct usb\_mouse {  char name[128];  char phys[64];  struct usb\_device \*usbdev;  struct input\_dev \*dev;  struct urb \*irq;  signed char \*data;  dma\_addr\_t data\_dma;  };  /\*функция обработки прерывания\*/  static void usb\_mouse\_irq(struct urb \*urb)  {  struct usb\_mouse \*mouse = urb->context;  signed char \*data = mouse->data;  struct input\_dev \*dev = mouse->dev;  int status;  switch (urb->status) {  case 0: // success  break;  case -ECONNRESET: // unlink  case -ENOENT:  case -ESHUTDOWN:  return;  // -EPIPE: should clear the halt  default: // error  goto resubmit;  }  /\*  input\_report\_key(dev, BTN\_LEFT, data[1] & 0x01);  input\_report\_key(dev, BTN\_RIGHT, data[1] & 0x02);  input\_report\_key(dev, BTN\_MIDDLE, data[1] & 0x04);  input\_report\_key(dev, BTN\_SIDE, data[1] & 0x08);  input\_report\_key(dev, BTN\_EXTRA, data[1] & 0x10);  input\_report\_rel(dev, REL\_X, data[2]); // In example data[1]  input\_report\_rel(dev, REL\_Y, data[3]); // In example data[2]  input\_report\_rel(dev, REL\_WHEEL, data[0]); // In example data[3]  //There is problem with y and wheel  input\_sync(dev);  \*/  status = usb\_submit\_urb (urb, GFP\_ATOMIC);  if(!(data[1] & 0x01) && !(data[1] & 0x02)) { // In example were data[0], in real data[1]  pr\_info("Other action!\n");  }  else if(data[1] & 0x01) {  pr\_info("Left mouse button clicked!\n");  btn\_left\_counter++;  }  else if(data[1] & 0x02) {  pr\_info("Right mouse button clicked!\n");  btn\_right\_counter++;  }  return;  resubmit:  status = usb\_submit\_urb (urb, GFP\_ATOMIC);  if (status)  printk(KERN\_ALERT "can't resubmit intr, %s-%s/input0, status %d", mouse->usbdev->bus->bus\_name, mouse->usbdev->devpath, status);  }  static int usb\_mouse\_open(struct input\_dev \*dev)  {  struct usb\_mouse \*mouse = input\_get\_drvdata(dev);  mouse->irq->dev = mouse->usbdev;  /\*  заполнение структуры urb.  Эта структура используется для передачи или приема данных от USB в асинхронном режиме.  \*/  if (usb\_submit\_urb(mouse->irq, GFP\_KERNEL))  return -EIO;  return 0;  }  static void usb\_mouse\_close(struct input\_dev \*dev)  {  struct usb\_mouse \*mouse = input\_get\_drvdata(dev);  printk(KERN\_INFO "Driver my usb mouse: usb\_mouse\_close(). btn\_right\_counter = %d, btn\_left\_counter = %d\n", btn\_right\_counter, btn\_left\_counter );  /\* освобождение структуры urb. \*/  usb\_kill\_urb(mouse->irq);  }  /\* подключение устройства к драйверу \*/  static int usb\_mouse\_probe(struct usb\_interface \*intf, const struct usb\_device\_id \*id)  {  struct usb\_device \*dev = interface\_to\_usbdev(intf);  struct usb\_host\_interface \*interface;  struct usb\_endpoint\_descriptor \*endpoint;  struct usb\_mouse \*mouse;  struct input\_dev \*input\_dev;  int pipe, maxp;    interface = intf->cur\_altsetting;  if (interface->desc.bNumEndpoints != 1)  return -ENODEV;  endpoint = &interface->endpoint[0].desc;  if (!usb\_endpoint\_is\_int\_in(endpoint))  return -ENODEV;  /\* usb\_rcvintpipe - определяет IN endpoint прерывания для USB устройства с заданным endpoint \*/  pipe = usb\_rcvintpipe(dev, endpoint->bEndpointAddress);  maxp = usb\_maxpacket(dev, pipe, usb\_pipeout(pipe));  mouse = kzalloc(sizeof(struct usb\_mouse), GFP\_KERNEL);  input\_dev = input\_allocate\_device();  if (!mouse || !input\_dev)  goto fail1;  /\* allocate dma-consistent buffer for URB\_NO\_xxx\_DMA\_MAP \*/  mouse->data = usb\_alloc\_coherent(dev, 8, GFP\_ATOMIC, &mouse->data\_dma);  if (!mouse->data)  goto fail1;  /\* creates a new urb (USB Request Block) for a USB driver to use \*/  mouse->irq = usb\_alloc\_urb(0, GFP\_KERNEL);  if (!mouse->irq)  goto fail2;  mouse->usbdev = dev;  mouse->dev = input\_dev;  input\_dev->name = mouse->name;  input\_dev->phys = mouse->phys;  usb\_to\_input\_id(dev, &input\_dev->id);  input\_dev->dev.parent = &intf->dev;  /\*  EV\_KEY type represents key press and release events,  EV\_REL type represents relative axis events (such as mouse movements)  REL\_WHEEL, REL\_HWHEEL: - These codes are used for vertical and horizontal scroll wheels, respectively.  \*/  input\_dev->evbit[0] = BIT\_MASK(EV\_KEY) | BIT\_MASK(EV\_REL);  input\_dev->keybit[BIT\_WORD(BTN\_MOUSE)] = BIT\_MASK(BTN\_LEFT) |  BIT\_MASK(BTN\_RIGHT) | BIT\_MASK(BTN\_MIDDLE);  input\_dev->relbit[0] = BIT\_MASK(REL\_X) | BIT\_MASK(REL\_Y);  input\_dev->keybit[BIT\_WORD(BTN\_MOUSE)] |= BIT\_MASK(BTN\_SIDE) |  BIT\_MASK(BTN\_EXTRA);  input\_dev->relbit[0] |= BIT\_MASK(REL\_WHEEL);  input\_set\_drvdata(input\_dev, mouse);  input\_dev->open = usb\_mouse\_open;  input\_dev->close = usb\_mouse\_close;  /\* Установка функции-обработчика прерывания \*/  usb\_fill\_int\_urb(mouse->irq, dev, pipe, mouse->data,  (maxp > 8 ? 8 : maxp),  usb\_mouse\_irq, mouse, endpoint->bInterval);  mouse->irq->transfer\_dma = mouse->data\_dma;  /\* URB\_NO\_TRANSFER\_DMA\_MAP - this bit is used for control urbs that have a DMA buffer already set up \*/  mouse->irq->transfer\_flags |= URB\_NO\_TRANSFER\_DMA\_MAP;  if (input\_register\_device(mouse->dev))  goto fail3;  /\* save our data pointer in this interface device \*/  usb\_set\_intfdata(intf, mouse);  return 0;  fail3:  usb\_free\_urb(mouse->irq);  fail2:  usb\_free\_coherent(dev, 8, mouse->data, mouse->data\_dma);  fail1:  input\_free\_device(input\_dev);  kfree(mouse);  return -1;  }  /\*отключение устройства от драйвера\*/  static void usb\_mouse\_disconnect(struct usb\_interface \*intf)  {  struct usb\_mouse \*mouse = usb\_get\_intfdata (intf);  usb\_set\_intfdata(intf, NULL);  if (mouse) {  usb\_kill\_urb(mouse->irq);  input\_unregister\_device(mouse->dev);  usb\_free\_urb(mouse->irq);  usb\_free\_coherent(interface\_to\_usbdev(intf), 8, mouse->data, mouse->data\_dma);  kfree(mouse);  }  }  static struct usb\_device\_id usb\_mouse\_id\_table [] = {  /\* USB HID interface subclass and protocol codes. \*/  { USB\_INTERFACE\_INFO(USB\_INTERFACE\_CLASS\_HID, USB\_INTERFACE\_SUBCLASS\_BOOT,  USB\_INTERFACE\_PROTOCOL\_MOUSE) },  { }  };  MODULE\_DEVICE\_TABLE (usb, usb\_mouse\_id\_table);  /\* имя драйвера должно быть установлено перед вызовом функции регитрации драйвера;  поле id содержит идентификатор шины (PCI, USB, ...), vendor ID и device ID устройства;  probe и disconnect - это callback-функции, вызываемые системой при  подключении и отключении USB-устройства. probe будет вызыван для  каждого устройства, если список id\_table пуст, или только для тех  устройств, которые соответствуют параметрам, перечисленным в списке.  \*/  static struct usb\_driver usb\_mouse\_driver = {  .name = "myusbmouse",  .probe = usb\_mouse\_probe,  .disconnect = usb\_mouse\_disconnect,  .id\_table = usb\_mouse\_id\_table,  };  static int \_\_init usb\_mouse\_init(void)  {  int retval = usb\_register(&usb\_mouse\_driver);  if (retval == 0)  printk(KERN\_INFO "myusbmouse init\n");  return retval;  }  static void \_\_exit usb\_mouse\_exit(void)  {  usb\_deregister(&usb\_mouse\_driver);  printk(KERN\_INFO "myusbmouse exit. btn\_right\_counter = %d, btn\_left\_counter = %d\n", btn\_right\_counter, btn\_left\_counter);  }  module\_init(usb\_mouse\_init);  module\_exit(usb\_mouse\_exit); |

Лог действий:

|  |
| --- |
| **// Для отключения драйвера usb мыши выгружаем модуль ядра usbhid**  nikita@nikita-K53SM:~/labs/2semester/lab7/3$ sudo rmmod usbhid  **// Соберем реализованный модуль ядра**  nikita@nikita-K53SM:~/labs/2semester/lab7/3$ make  make -C /lib/modules/4.4.0-75-generic/build M=/home/nikita/labs/2semester/lab7/3 modules  make[1]: вход в каталог «/usr/src/linux-headers-4.4.0-75-generic»  Building modules, stage 2.  MODPOST 1 modules  make[1]: выход из каталога «/usr/src/linux-headers-4.4.0-75-generic»  **// Загрузка модуля ядра и просмотр сообщений о нажатых кнопках**  nikita@nikita-K53SM:~/labs/2semester/lab7/3$ sudo insmod myusbmouse.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/3$ dmesg | tail  [14583.177942] myusbmouse init  [14583.214541] Driver my usb mouse: usb\_mouse\_close(). btn\_right\_counter = 0, btn\_left\_counter = 0  [14585.463234] Other action!  [14585.471232] Other action!  [14585.479167] Right mouse button clicked!  [14585.607232] Other action!  [14586.063230] Other action!  [14586.071236] Other action!  [14587.571224] Left mouse button clicked!  [14587.651224] Other action!  **// Выгрузка модуля из ядра**  nikita@nikita-K53SM:~/labs/2semester/lab7/3$ sudo rmmod myusbmouse.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/3$ dmesg | tail  [14585.471232] Other action!  [14585.479167] Right mouse button clicked!  [14585.607232] Other action!  [14586.063230] Other action!  [14586.071236] Other action!  [14587.571224] Left mouse button clicked!  [14587.651224] Other action!  [14607.516191] usbcore: deregistering interface driver myusbmouse  [14607.538705] Driver my usb mouse: usb\_mouse\_close(). btn\_right\_counter = 1, btn\_left\_counter = 1  [14607.562503] myusbmouse exit. **btn\_right\_counter = 1, btn\_left\_counter = 1**  **// Вернем в систему исходный модуль ядра usbhid**  nikita@nikita-K53SM:~/labs/2semester/lab7/3$ sudo modprobe usbhid |

После выгрузки модуля ядра usbhid usb мышь перестает работать. Далее собираем реализованный модуль ядра командой make и загружаем его. После нажатия левой и правой кнопки мыши в системном журнале отображаются соответствующие сообщения. После выгрузки модуля в системный журнал записывается информация о выгрузке драйвера, а также количество нажатий на правую и левую кнопку мыши. После эксперимента в систему был загружен исходный модуль ядра usbhid.

**4. Создание многослойного драйвера**

Многослойный драйвер был построен на основе драйвера символьного устройства, модуля, обращающегося к порту клавиатуры, и модуля ядра usb-мыши, который отслеживает нажатие правой и левой кнопки мыши, а также подсчитывает количество нажатий. При чтении символьного устройства пользователь теперь получает также информацию о количестве прерываний клавиатуры, нажатий на кнопку Esc, а также на правую и левую кнопку мыши. Чтобы драйвер символьного устройства имел доступ к функциям других модулей, все модули нужно собирать вместе. Для того чтобы функция одного модуля была доступна другому модулю необходимо использовать перед её объявлением ключевое слово **extern**. **EXPORT\_SYMBOL** используется для экспортирования точек входа функций **getIrqCounter, getEscCounter, getBtnRightCounter, getBtnLeftCounter**. Были созданы файлы mykeyboard\_workqueue.h и myusbmouse.h, в которых объявлены функции, которые должны быть доступны модулю символьного устройства.

|  |
| --- |
| **// mykeyboard\_workqueue.h**  extern int getIrqCounter( void );  extern int getEscCounter( void );  **// Изменения в mykeyboard\_workqueue.c**  …  #include "mykeyboard\_workqueue.h"  …  int getIrqCounter( void )  {  return irq\_counter;  }  EXPORT\_SYMBOL( getIrqCounter );  int getEscCounter( void )  {  return esc\_counter;  }  EXPORT\_SYMBOL( getEscCounter );  …  **// myusbmouse.h**  extern int getBtnRightCounter( void );  extern int getBtnLeftCounter( void );  **// Изменения в myusbmouse.c**  ...  #include "myusbmouse.h"  ...  int getBtnRightCounter( void )  {  return btn\_right\_counter;  }  EXPORT\_SYMBOL( getBtnRightCounter );  int getBtnLeftCounter( void )  {  return btn\_left\_counter;  }  EXPORT\_SYMBOL( getBtnLeftCounter );  …  **// Изменения в chardev.c**  #include "mykeyboard\_workqueue.h"  #include "myusbmouse.h"  …  #define BUF\_LEN 500 /\* Максимальная длина сообщения \*/  …  if (length > 0) {  sprintf(Message, "irq\_counter = %d, esc\_counter = %d, btn\_right\_counter = %d, btn\_left\_counter = %d\n", getIrqCounter(), getEscCounter(), getBtnRightCounter(), getBtnLeftCounter());  Message\_Ptr = Message;  }  /\* Помещение данных в буфер о количестве прерываний с клавиатуры и нажати кнопки Esc \*/  while (length && \*Message\_Ptr) {  put\_user(\*(Message\_Ptr++), buffer++);  length--;  bytes\_read++;  }  ... |

Содержимое **Makefile**:

|  |
| --- |
| SRCS = chardev.c mykeyboard\_workqueue.c myusbmouse.c  OBJS = $(SRCS:.c=.o)  obj-m += $(OBJS)  EXTRA\_CFLAGS = -O3  all:  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules  clean:  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean |

Лог действий:

|  |
| --- |
| **//1. Получение файлов с расширением .ko**  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ make  make -C /lib/modules/4.4.0-75-generic/build M=/home/nikita/labs/2semester/lab7/4 modules  make[1]: вход в каталог «/usr/src/linux-headers-4.4.0-75-generic»  Building modules, stage 2.  MODPOST 3 modules  make[1]: выход из каталога «/usr/src/linux-headers-4.4.0-75-generic»  **//2. Для отключения драйвера usb мыши выгружаем модуль ядра usbhid**  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo rmmod usbhid  **//3. Загрузка модулей в ядро системы Linux**  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo insmod myusbmouse.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo insmod mykeyboard\_workqueue.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo insmod chardev.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ dmesg | grep mknod  [18169.000365] 'mknod /dev/chardev c 244 0'.  **//4. Создание файла символьного устройства**  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo mknod /dev/chardev c 244 0  **//5. Изменение прав доступа к файлу для поддержки возможности записи в файл**  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo chmod 666 /dev/chardev  **//6. Запись в файл устройства**  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ echo 'I say hello!' > /dev/chardev  **//7. Чтение файла устройства**  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ cat /dev/chardev  **I say hello!**  **I already told you 1 times HELLO!**  **irq\_counter = 187, esc\_counter = 1, btn\_right\_counter = 2, btn\_left\_counter = 2**  **//8. Выгрузка модулей из ядра и удаление файла устройства**  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo rmmod chardev.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo rm -i /dev/chardev  rm: удалить символьный специальный файл '/dev/chardev'? y  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo rmmod mykeyboard\_workqueue.ko  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo rmmod myusbmouse.ko  **//9. Вернем в систему исходный модуль ядра usbhid**  nikita@nikita-K53SM:~/labs/2semester/lab7/4$ sudo modprobe usbhid |

Драйвер позволяет записывать информацию в символьное устройство, а также читать файл устройства. При чтении также отображается информация о количестве прерываний клавиатуры, нажатий на кнопку Esc, а также на правую и левую кнопку мыши.

**Вывод**

Был написан драйвер символьного устройства, модуль, обращающийся к порту клавиатуры, и модуль ядра usb-мыши, который отслеживает нажатие правой и левой кнопки мыши, а также подсчитывает количество нажатий. Потом отдельные модули были объединены в многослойный драйвер, который позволяет при чтении символьного устройства получать информации о количестве нажатий на правую и левую кнопку мыши, клавишу Esc, а также о количестве прерываний на клавиатуре.

Модуль, обращающийся к порту клавиатуры, был реализован в двух вариантах. В первом варианте вся обработка находится в верхней половине. Во втором варианте используется очередь отложенных действий (нижняя половина), которая записывает в буфер сообщений ядра информацию о количестве прерываний клавиатуры.

Были проведены эксперименты с обработкой событий при наличии аппаратных прерываний. Первая программа подсчитывала количество прерываний от клавиатуры.

Следующий эксперимент позволил пронаблюдать изменение системного счетчика. Тасклеты и очереди отложенных действий представляют две различные схемы реализации отложенных работ в современном Linux, которые переносят работы из верхних половин в нижние половины драйверов. В тасклетах реализуется механизм с низкой латентностью, который является простым и ясным, а очереди работ имеют более гибкий и развитый API, который позволяет обслуживать несколько отложенных действий в порядке очередей. В каждой схеме откладывание (планирование) последующей работы выполняется из контекста прерывания, но только тасклеты выполняют запуск автоматически в стиле «работа до полного завершения», тогда как очереди отложенных действий разрешают функциям-обработчикам переходить в блокированные состояния. В этом состоит главное принципиальное отличие: рабочая функция тасклета не может блокироваться.

**Список использованных источников**

1. Основные сведения о механизме прерываний. Работа системы прерываний в реальном режиме. [Электронный ресурс] // URL: <http://mf.grsu.by/UchProc/livak/b_org/oal_10.htm>. (Дата обращения: 02.04.2017).
2. Общая модель обработки прерываний. [Электронный ресурс] // URL: <http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-25.html>. (Дата обращения: 02.04.2017).
3. Цилюрик О.И. “Модули ядра Linux”. [Электронный ресурс] // URL: <http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-29.html> (Дата обращения: 02.04.2017).
4. Драйверы для USB-устройств — это просто. [Электронный ресурс] // URL: [http://wiki.metrotek.spb.ru/wiki/%D0%94%D1%80%D0%B0%D0%B9%D0%B2%D0%B5%D1%80%D1%8B\_%D0%B4%D0%BB%D1%8F\_USB-%D1%83%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2\_%E2%80%94\_%D1%8D%D1%82%D0%BE\_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D0%BE](http://wiki.metrotek.spb.ru/wiki/Драйверы_для_USB-устройств_—_это_просто) (Дата обращения: 02.05.2017).
5. http://www.makelinux.net/ldd3/chp-13-sect-4
6. <http://www.makelinux.net/ldd3/chp-13-sect-3>
7. Пример кода. [Электронный ресурс] // URL: <https://github.com/spotify/linux/blob/master/drivers/hid/usbhid/usbmouse.c> (Дата обращения: 02.05.2017).
8. Отладка и тестирование модулей ядра: Часть 75. Отладка в ядре. Общие положения. [Электронный ресурс] // URL: <https://www.ibm.com/developerworks/ru/library/l-linux_kernel_75/> (Дата обращения: 02.05.2017).