Содержание

Введение

Для решения широкого круга задач зачастую требуется получить информацию о функционировании какого-либо периферийного устройства компьютера, т. е. произвести мониторинг его производительности.

В настоящее время активно используются устройства, подключающиеся к компьютеру посредством USB-порта. USB (*Universal Serial Bus*— «универсальная последовательная шина»)— последовательный интерфейса передачи данных для среднескоростных и низкоскоростных периферийных устройств в вычислительной технике. Благодаря встроенным линиям питания USB позволяет подключать периферийные устройства без собственного источника питания.

Соответственно, возникает необходимость в создании ПО, которое бы отслеживало работу устройства, подключенного через USB-порт, к примеру подсчитывало количество переданных/принятых пакетов и их размер, что позволило бы судить о скорости обмена данными между внешним устройством и компьютером.

1. Аналитический раздел

1.1 Постановка задачи

Разработать модуль ядра для компьютера, работающего под ОС Linux, позволяющий подсчитывать количество пакетов, отправленных с устройства/на устройство, а так же их размер. Данные подсчитываются для всех устройств, подключенные в данных момент по usb-портам.

1.2 Загружаемые модули ядра

Одна из важных особенностей ядра Linux – это способность расширять собственную функциональность непосредственно в период выполнения. Каждый фрагмент исполняемого кода, который может быть добавлен в ядро во время его работы, называется модулем ядра. Без загружаемых модулей ядра, операционные системы должны были бы включать всю возможную функциональность в базовом ядре. Значительная часть кода не используется и лишь занимает память. Каждый раз, когда пользователю необходима новая функциональность, еще не включенная в базовом ядре, требуется полная перекомпиляция базового ядра и перезагрузка. Использование подгружаемых модулей значительно упрощает изменение функциональности ядра и не требует ни полной перекомпиляции (модуль часто может быть собран отдельно от ядра или поставлен в предкомпилированном виде) ни перезагрузок.

Модуль может быть загружен в ядро с помощью программы insmod (вызывающей функции *create_module() / init_module()*), и выгружен с помощью rmmod (вызывающего *delete_module()*). В данной работе реализует именно такой динамически загружаемый модуль. Для сборки модуля необходим специальный скрипт - Makefile. Его содержание для сборки различных модулей не сильно меняется.

Пример Makefile:

```
EXTRA_CFLAGS += -Wno-strict-prototypes -Werror
KERNEL_DIR ?= /lib/modules/$(shell uname -r)/build
KMAKE := make -C $(KERNEL_DIR) M=$(PWD)
```

```
obj-m += myMod.o

all:
	$(KMAKE) modules

clean:
	$(KMAKE) clean
```

1.2.1 Структура загружаемого модуля

Для начала можно отметить, что модуль являтеся объектным файлом формата ELF:

```
➜  doIt  file myMod.ko
myMod.ko: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV),
BuildID[sha1]=0xb4e41bdb1b9252befe6055f45643b713ed23d75e, not stripped
```

Структура секций объектного файла модуля такова (оставлены наиболее важые):

```
➜  doIt  objdump -h myMod.ko

myMod.ko:     формат файла elf64-x86-64

Разделы:
Инд Имя          Размер    VMA              LMA              Файл     Вырав
 ...
  1 .text         000002b4  0000000000000000  0000000000000000  00000070  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  2 .init.text    00000105  0000000000000000  0000000000000000  00000324  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  3 .exit.text    0000007e  0000000000000000  0000000000000000  00000429  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
...
  7 .modinfo      00000072  0000000000000000  0000000000000000  000005eb  2**0
```

CONTENTS, ALLOC, LOAD, READONLY, DATA

...

*10 .data 000000c0 0000000000000000 0000000000000000 00000ce0 2**5*

CONTENTS, ALLOC, LOAD, RELOC, DATA

...

*12 .bss 00000038 0000000000000000 0000000000000000 00001000 2**3*

ALLOC

...

Здесь секции:

- .text — код модуля (инструкуции);
- .init.text,.exit.text — код инициализации модуля и завершения, соответственно;
- .modinfo — текст макросов модуля;
- .data — инициализированные данные;
- .bss — не инициализированные данные (Block Started Symbol);

При сборке модуля сначала создается объектный файл с форматом *.o, а затем *.ko.

➜ *doIt modinfo myMod.o*

filename: myMod.o

license: GPL

➜ *doIt modinfo myMod.ko*

filename: myMod.ko

license: GPL

srcversion: FAE3E39FB029A64301E1923

depends:

vermagic: 3.8.6-030806-generic SMP mod_unload modversions

Легко видеть, что при сборке к файлу модуля добавлено несколько внешних имен, значения которых используются системой для контроля возможности корректной загрузки модуля.

1.3 Устройство USB

Нет смысла говорить о таких очевидных вещах, как широкое распространение USB, высокая скорость обмена по USB, возможность горячего подключения устройств. Сейчас уже каждый пользователь ПК так или иначе оценил преимущества и плюсы USB.
Поэтому сразу перейдем к менее очевидным вещам - к тому, как USB устроен внутри.

1.3.1 Концепция

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.

Есть только одно исключение: после того, как хост перевел устройство в suspend-режим, устройство может посылать запрос remote wakeup. Во всех остальных случаях хост формирует запросы, а устройства отвечают на них.

Система USB разделяется на три логических уровня с определенными правилами взаимодействия. Устройство USB содержит интерфейсную, логическую и функциональную части. Хост тоже делится на три части: интерфейсную, системную и программное обеспечение. Каждая часть отвечает только за определенный круг задач. Логическое и реальное взаимодействие между ними показано на рисунке :

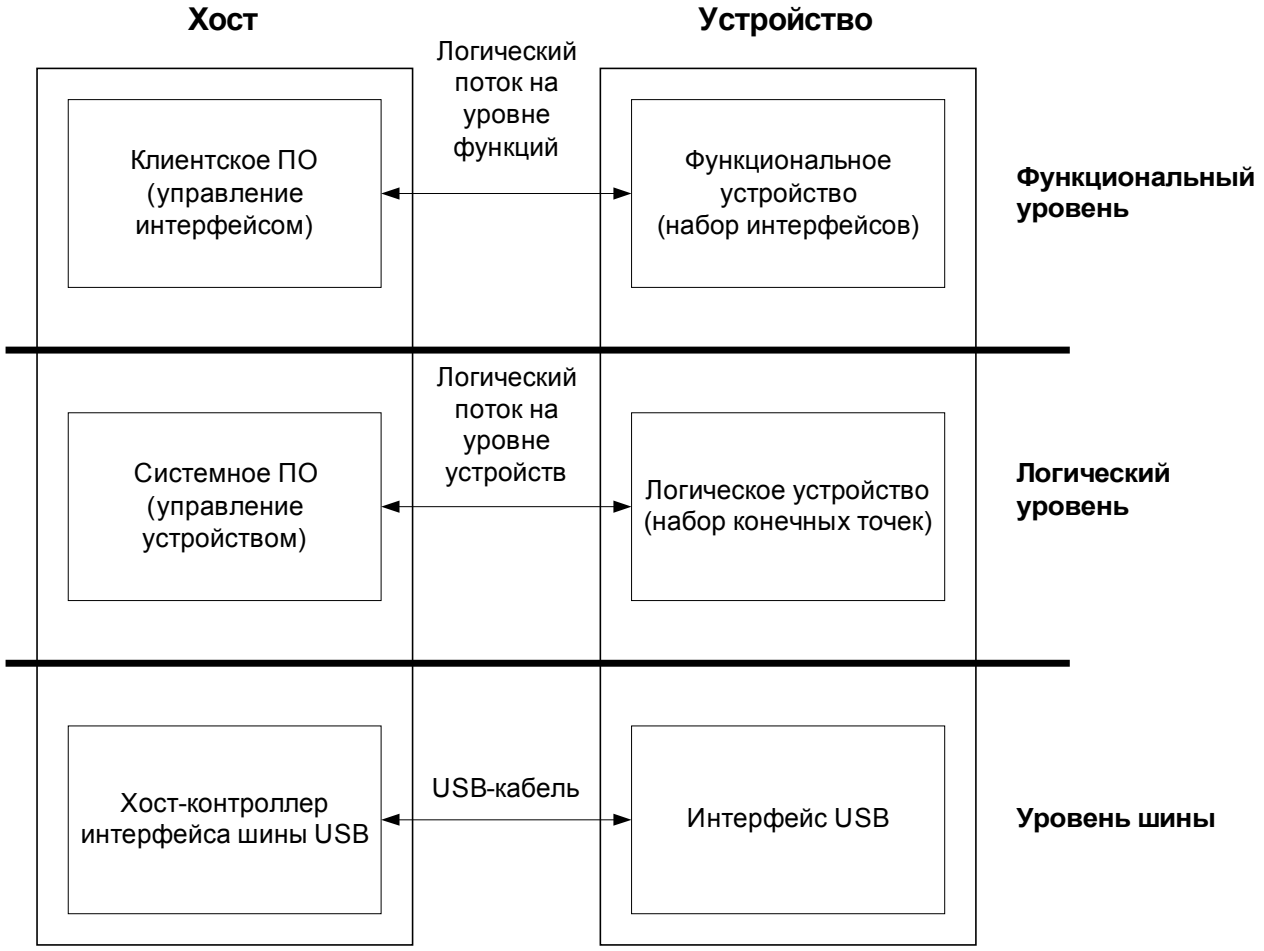


Рис. 1.3.1 Взаимодействие компонентов USB.

Таким образом, операция обмена данными между прикладной программой и шиной USB выполняется путем передачи буферов памяти через следующие уровни:

- уровень клиентского программного обеспечения в хосте – обычно представляется драйвером устройства USB, обеспечивает взаимодействие пользователя с операционной системой с одной стороны и системным драйвером с другой;
- уровень системного программного обеспечения USB в хосте (USBD, Universal Serial Bus Driver) – управляет нумерацией устройств на шине, управляет распределением пропускной способности шины и мощности питания, обрабатывает запросы пользовательских драйверов;
- хост-контроллер интерфейса шины USB (HCD, Host Controller Driver) – преобразует запросы ввода/вывода в структуры данных, по которым хост-контроллер выполняет физические транзакции, работает с регистрами хост-контроллера.

1.3.2 Блоки запроса USB(URB)

Код USB в ядре Linux взаимодействует со всеми устройствами USB помощью так называемых URB(USB request block, блок запроса USB). Этот блок запроса описывается структурой ***struct urb*** и может быть найден в файле ***include/linux/usb.h***.

URB используется для передачи или приема данных в или из заданной конечной точки USB на заданное USB устройство в асинхронном режиме. В зависимости от потребностей, драйвер USB устройства может выделить для одной конечной точке много urb-ов или может повторно использовать один urb для множества разных конечных точек. Каждая конечная точка в устройстве может обрабатывать очередь urb-ов, так что перед тем, как очередь опустеет, к одной конечной точке может быть отправлено множество urb-ов.
Типичный жизненный цикл urb выглядит следующим образом:

- Создание драйвером USB;
- Назначение в определенную конечную точку заданного USB устройства;
- Передача драйвером USB-устройства в USB-ядро;
- Передача USB-ядром в заданный драйвер контроллера USB-узла для указанного устройств;
- Обработка драйвером контроллера USB-узла, который выполняет передачу по USB в устройство;
- После завершения работы с URB, драйвер контроллера USB узла уведомляет драйвер USB устройства

URB также могут быть отменены в любое время драйвером, который передал URB, или ядром USB, если устройство удалено из системы. URB создаются динамически и содержат внутренний счетчик ссылок, что позволяет им быть автоматически освобождаемыми, когда последний пользователь URB отпускает его.

Полями структуры **struct urb**, которые имеют значение для драйвера USB устройства являются:
struct usb_device *dev
Указатель на **struct usb_device**, в которую послан этот urb. Эта переменная должна быть проинициализирована драйвером USB перед тем, как urb может быть отправлен в ядро USB.

unsigned int pipe

Информация конечной точки для указанной **struct usb_device**, которую этот usb будет передавать.

unsigned int transfer_flags

Эта переменная может быть установлена в несколько различных битовых значений, в зависимости от того, что драйвер USB хочет, чтобы происходило с urb.
Например URB_DIR_OUT, URB_DIR_IN

void *transfer_buffer

Указатель на буфер, который будет использоваться при передаче данных в устройство (для ВЫХОДНОГО urb) или при получении данных из устройства (для ВХОДНОГО urb). Для того, чтобы хост-контроллер правильно получал доступ к этому буферу, он должен быть создан вызовом ***kmalloc***, а не на стеке или статически. Для управляющих оконечных точек этот буфер для стадии передачи данных.

dma_addr_t transfer_dma

Буфер для использования для передачи данных в USB устройство с помощью DMA.

int transfer_buffer_length

Длина буфера, на который указывает переменная **transfer_buffer** или **transfer_dma** (только одна из них может быть использована для urb). Если она 0, USB ядром никакие буферы передачи не используются.
Для ВЫХОДНОЙ конечной точки, если максимальный размер конечной точки меньше значения, указанного в этой переменной, передача в USB устройство разбивается на мелкие куски, чтобы правильно передать данные. Это большая передача происходит в последовательных кадрах USB. Гораздо быстрее поместить большой блок данных в один urb и предоставить контроллеру USB узла разделить его на мелкие куски, чем отправить маленькие буферы в последовательно.

unsigned char *setup_packet

Указатель на установочный пакет для управляющего urb. Он передается перед данными в буфере передачи. Эта переменная действительна только для управляющих urb.

dma_addr_t setup_dma

DMA буфер для установочного пакета для управляющего urb. Он передается перед данными в обычном буфере передачи. Эта переменная действительна только для управляющих urb-ов.

usb_complete_t complete

Указатель на завершающую функцию обработки, которая вызывается USB ядром, когда urb полностью передан или при возникновении ошибки с urb. В этой функции драйвер USB может проверить urb, освободить его, или использовать повторно для другой передачи.

void *context

Указатель на данные blob (Binary Large Object, большой двоичный объект), который может быть установлены USB драйвером. Он может быть использован в завершающем обработчике, когда urb возвращается в драйвер. Смотрите следующий раздел для более подробной информации об этой переменной.

int actual_length

После завершения urb эта переменная равна фактической длине данных или отправленных urb или полученных urb . Для ВХОДНЫХ urb, она должна быть использована вместо переменной **transfer_buffer_length**, поскольку полученные данные могут быть меньше, чем размер всего буфера.

int status

После завершения urb или его обработки USB ядром, эта переменная содержит текущий статус urb. Единственным временем, когда USB драйвер может безопасно получить доступ к этой переменной, является функция обработчика завершения urb. Это ограничение является защитой от состояний гонок, которые происходят в то время, как urb обрабатывается USB ядром. Для изохронных urb успешное значение (0) в этой переменной указывает лишь, был ли urb отсоединен. Для получения подробного статуса изохронных urb должны быть проверены переменные **iso_frame_desc**.

Допустимые значения для этой переменной включают:

0

Передача urb была успешной.

-ENOENT

urb был остановлен вызовом ***usb_kill_urb***.

-ECONNRESET

urb были отсоединен вызовом ***usb_unlink_urb*** и переменная **transfer_flags**в urb была установлена в **URB_ASYNC_UNLINK**.

-EINPROGRESS

urb все еще обрабатывается контроллерами узлов USB. Если ваш драйвер когда-нибудь увидит это значение, это является ошибкой в вашем драйвере.

-EILSEQ

Было несоответствие контрольной суммы (CRC) при передаче urb.

-EPIPE

Оконечная точка сейчас застряла. Если данная оконечная точка не является управляющей оконечной точкой, эта ошибка может быть сброшена вызовом функции ***usb_clear_halt***.

-ECOMM

Данные в течение передачи были получены быстрее, чем они могли быть записаны в память системы. Эта ошибка случается только с ВХОДНЫМИ urb.

-ENOSR

Данные не могут быть получены из системной памяти при передаче достаточно быстро, чтобы сохранять запрошенную скорость передачи данных USB. Эта ошибка случается только с ВЫХОДНЫМИ urb.

-EOVERFLOW

С urb-ом произошла ошибка "помеха". Ошибка "помеха" происходит, когда оконечная точка принимает больше информации, чем указанный максимальный размер пакета оконечной точки.

-EREMOTEIO

Возникает только если в переменной urb **transfer_flags** установлен флаг **URB_SHORT_NOT_OK** и означает, что весь объем данных, запрошенный этим urb-ом, не был принят.

-ENODEV

USB устройство является теперь отключенным от системы.

-EXDEV

Происходит только для изохронного urb и означает, что передача была выполнена лишь частично. Для того, чтобы определить, что было передано, драйвер должен посмотреть на статус отдельного фрейма.

-EINVAL

С urb-ом произошло что-то очень плохое. Документация USB ядра описывает, что это значение означает:

> ISO madness, if this happens: Log off and go home
> ISO обезумело,если это происходит: завершите работу и идите домой

Это также может произойти, если неправильно установлен какой-то параметр в структуре urb или если в USB ядро urb поместил неправильный параметр функции в вызове ***usb_submit_urb***.

-ESHUTDOWN

Были серьезные ошибки в драйвере контроллера USB узла; теперь он запрещен или устройство было отключено от системы, а urb был получен после удаления устройства. Она может также возникнуть, если во время помещения urb в устройство для данного устройства была изменена конфигурация.

int start_frame

Устанавливает или возвращает для использования номер начального фрейма для изохронной передачи.

int interval

Интервал, с которым собираются urb. Это справедливо только для urb прерывания или изохронных. Единицы значения существенно отличаются в зависимости от скорости устройства. Для низкоскоростных и полноскоростных устройств единицами являются фреймы, которые эквивалентны миллисекундам. Для высокоскоростных устройств единицами являются микрофреймы, что эквивалентно единицам в 1/8 миллисекунды. Эта значение должно быть установлено драйвером USB для изохронных urb или urb прерывания до того, как urb посылается в USB ядро.

int number_of_packets

Имеет смысл только для изохронных urb и определяет число изохронных буферов передачи, которые должен обработать этот urb. Эта величина должна быть установлена драйвером USB для изохронных urb передачи до того, как urb посылается в USB ядро.

int error_count

Устанавливается USB ядром только для изохронных urb после их завершения. Она определяет число изохронных передач, которые сообщили о любых ошибках.

struct usb_iso_packet_descriptor iso_frame_desc[0]

Имеет смысл только для изохронных urb. Эта переменная представляет собой массив из структур **struct usb_iso_packet_descriptor**, которые составляют этот urb. Такая структура позволяет сразу одним urb определить число изохронных передач. Она также используется для сбора статуса передачи каждой отдельной передачи.

struct usb_iso_packet_descriptor состоит из следующих полей:

unsigned int offset
Смещение в буфере передачи (начиная с 0 для первого байта), где расположены данные этого пакета.

unsigned int length
Размер буфера передачи для этого пакета.

unsigned int actual_length
Длина данных, полученных в буфере передачи для этого изохронного пакета.

unsigned int status
Статус отдельной изохронной передачи этого пакета. Он может иметь такие же возвращаемые значения, как основная переменная статуса структуры **struct urb**.

1.3.3 Направление

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.

1.3.4 Классификация пакетов в USB

По USB может передаваться несколько типов пакетов:

1. Token - запрос, содержит управляющую информацию: направление операции (IN, OUT), номер endpoint
2. Data - пакет данных
3. Handshake - служебные пакеты, могут содержать подтверждение (ACK), сообщение об ошибке, отказ (NACK)
4. Special - служебные пакеты, такие как PING

Пример: отсылка данных устройству

Чтобы отослать данные устройству, хост посылает пакет Token "OUT", затем пакет Data. Если устройство готово обработать принятые данные, оно отсылает пакет Handshake "ACK", подтверждающий транзакцию. Если оно занято, оно отсылает отказ - Handshake "NACK". Если произошла какая-то ошибка, то устройство может не отсылать Handshake.

Пример: отсылка данных хосту

Как уже говорилось, устройство самостоятельно никогда не отсылает данные. Только по запросу. Чтобы принять данные, хост посылает пакет Handshake "IN". Устройство по запросу может отослать пакет Data, а затем Handshake "ACK". Либо может отослать Handshake "NACK", не посылая Data.

1.3.5 Типы передачи данных

Спецификация USB определяет 4 типа потоков данных:

1. **bulk transfer** - предназначен для пакетной передачи данных с размером пакетов 8, 16, 32, 64 для USB 1.1 и 512 для USB 2.0. Используется алгоритм перепосылки (в случае возникновения ошибок), а управление потоком осуществляется с использованием handshake пакетов, поэтому данный тип является достоверным. Поддерживаются оба направления - IN и OUT.

2. **control transfer** - предназначен для конфигурирования и управления устройством. Также, как и в bulk, используются алгоритмы подтверждения и перепосылки, поэтому этот тип обеспечивает гарантированный обмен данными. Направления - IN (status) и OUT(setup, control).

3. **interrupt transfer** - похож на bulk. Размер пакета - от 1 до 64 байт для USB 1.1 и до 1024 байт для USB 2.0. Этот тип гарантирует, что устройство будет опрашиваться (то есть хост будет отсылать ему token "IN") хостом с заданным интервалом. Направление - IN.

4. **isochronous transfer** - предназначен для передачи данных без управления потоком (без подтверждений). Область применения - аудио-потоки, видео-потоки. Размер пакета - до 1023 байт для USB 1.1 и до 1024 байт для USB 2.0. Предусмотрен контроль ошибок (на приемной стороне) по CRC16. Направления - IN и OUT.

1.3.6 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.

Например, допустим, что хост хочет прочитать пакет данных из EP4 (4го endpoint'а) устройства, пользуясь типом "bulk transfer". Он отсылает пакет token "in", в котором указывает адрес endpoint'а. Соответствующий источник в устройстве отсылает пакет данных хосту.

Аналогично происходит передача пакета данных.

1.3.6.1 Endpoint No.0

EP0 имеет особое значение для USB. Это Control EP. Он должен быть в каждом USB-устройстве. Этот EP использует token "setup", чтобы сигнализировать, что данные, отправляемые после него, предназначены для управления устройством.
Используя этот EP0, хост может передавать setup-пакет длиной 8 байт и данные, которые следуют за этим пакетом. Во многих случаях может хватать передачи только setup-пакета. Однако устройство может использовать и передачу данных по EP0, например для смены прошивок компонентов устройства, или получения расширенной информации об устройстве.
Рассмотрим немного подробнее setup-пакет.

EP0: setup-пакет

Содержимое setup-пакета представлено в таблице:

Байт (No.)	Имя	Назначение
0	bmRequestType	Поле для указания типа запроса, направления, получателя
1	bRequest	идентификатор запроса
2	wValueL	16-битное значение wValue, зависит от запроса.
3	wValueH	
4	wIndexL	16-битное значение wIndex, зависит от запроса.
5	wIndexH	
6	wLengthL	количество байт, отсылаемых после setup-пакета.
7	wLengthH	

Как видно из таблицы, setup-пакет содержит 5 полей. bmRequestType и bRequest определяют запрос, а wValue, wIndex и wLength - его свойства.

Спецификация USB резервирует диапазон значений bRequest под стандартные запросы. Каждое устройство обязано отвечать на все стандартные запросы. В следующей таблице приведены только несколько стандартных запросов, с которыми мы будем сталкиваться дальше.

bRequest	Имя	Описание
0x05	Set Address	установка уникального адреса устройства в системе
0x06	Get Descriptor	получение информации об устройстве. Тип информации зависит от поля wValue.

1.3.7 Распознавание устройства

Как происходит распознавание устройства, только что подключившегося к системе? Уже упоминалось, что каждое устройство обязано обеспечить доступ к EP0. Но кроме этого, оно еще должно отвечать на запросы, указанные в спецификации USB для EP0. Пользуясь этими запросами и происходит распознавание устройства в системе.

Алгоритм детектирования нового устройства следующий:

1. Хост отсылает setup-пакет "Get Descriptor" (wValue = "device");
2. Хост получает идентифицирующую информацию об устройстве;
3. Хост отсылает setup-пакет "Set address", после чего устройство; получает уникальный адрес в системе;
4. Хост отсылает остальные setup-пакеты "Get Descriptor" и получает дополнительную информацию об устройстве: количество EP, требования к питанию, и т.п.

1.3.8 Поддержка USB в ядре Linux

Программный интерфейс для взаимодействия с USB устройствами в ядре Linux очень прост. За простым интерфейсом скрываются все алгоритмы отсылки запросов, отслеживания подтверждений, контроля ошибок и т.п.

В ядре файлы программ располагаются в drivers/usb/, а заголовочные файлы - в

include/linux/. Информации, представленной в этих директориях, достаточно, чтобы самостоятельно написать драйвер для любого USB-устройства.

Драйвер, взаимодействующий с USB-устройством(-ами), как правило,выполняет следующие действия:

1. Регистрация/выгрузка драйвера;
2. Регистрация/удаление устройства;
3. Обмен данными: управляющий и информационный.

1.4 Usbmon

Для написания моего модуля оказалось необходимым исследовать уже имеющийся в ядре модуль — usbmon.

Модуль этот появился в ядре Linux с версии 2.6.11.

«usbmon» (строчными буквами) — это модуль ядра Linux, позволяющий собирать информацию(логи) о вводе/выводе. Usbmon сообщает о запросах драйверов периферийных устройств к драйверам контроллера хоста (Host controller drivers, далее HDC). У usbmon есть интерфейс, который предоставляет логи в текстовом формате, что позволяет пользователю прочесть их даже без специальных программ.

На самом верхнем уровне архитектуры, usbmon несложен. Он состоит из циклических буферов, наполняемых обработчиками в стеке USB. Каждый вызов помещает событие в буфер. Отсюда эти события берутся для дальнейшей обработки или уже представляются пользователю.

Запросы на ввод/вывод в стеке USB представлены urb, о которых уже говорилось ранее. Драйвер периферийного устройства, например usb-storage, инициализирует и подтверждает urb вызовом к стеку ядра USB. Ядро отсылает urb к HCD. Когда ввод/вывод окончен, HCD вызывает специальную функцию обратного вызова (callback), чтобы уведомить ядро и запрашивающий драйвер.

2. Конструкторский раздел

2.1 Структура разрабатываемого модуля

Код модуля содержится в одном файле — MyMonitor.c.

2.2. Инициализация и выгрузка модуля

Инициализацию модуля выполняет функция *static int __init mon_init(void).* Она вызывается при загрузке модуля в контексте процесса, вызвавшего *module_init()* (системный вызов загрузки модуля), и выполняет следующие действия:

1) Инициализирует необходимые для работы модуля структуры;
2) Проверяет, не занято ли устройство;
3) Если возможны не занято, вызывает функции urb_submit, urb_submit_error, urb_complete, инициализирует новый поток для записи в буфер сообщений ядра сведений о пакетах;
4) Создает директорию */sys/kernel/my_kobject* и файлы *in* и *out* в ней;
5) Для каждого порта usb-шины (ubus) в списке usb_bus_list выставляет значение monitored в 1(то есть, порт шины просмотрена);

Создание файлов происходит в последнюю очередь для того, чтобы пользовательские приложения не могли обратиться к драйверу до завершения инициализации.

Выгрузку модуля выполняет функция *static void __exit mon_exit(void),* вызванная в контексте процесса, сделавшего exit_module() (системный вызов для выгрузки модуля).
Выполняемые им действия:
1) Освобождает память;
2) Для всех шин выставляет значение monitored в 0;
3) Останавливает поток записи в буфер сообщений ядра;
4) Удаляет директорию */sys/kernel/my_kobject.*

2.3. Взаимодействие с пользовательскими приложениями

Для взаимодействия с пользовательскими приложениями драйвер использует файловую систему sys*fs* – псевдо-файловую систему, предоставляющую различную информацию о системе. Любой драйвер может добавлять в ее иерархию свои файлы и папки для передачи пользовательским программам различной информации.

Файловая система sysfs в Linux отличается от procfs тем, что предоставляет детализированную информацию о работе ядра пользователю (например, параметры устройств и загруженных модулей). Информация строго организована и обычно форматируются простом ASCII тексте, что делает ее очень доступной для пользователей и приложений.

Базовым понятием модели представления устройств являются объекты *struct kobjec*t(определяется в файле <linux/kobject.h>). Тип **struct kobject** по смыслу аналогичен абстрактному базовому классу Object в объектно-ориентированных языков программирования, как C# и Java. Этот тип определяет общую функциональность, такую как счетчик ссылок, имя, указатель на родительский объект, что позволяет создавать объектную иерархию.

Kobject-ы являются механизмом, стоящим за виртуальной файловой системой sysfs. Для каждого каталога, находящегося в sysfs, существует kobject, скрывающийся где-то в ядре. Каждый kobject интересен также экспортом одного или более атрибутов, которые появляются в каталоге sysfs kobject-а как файлы, содержащие генерируемую ядром информацию.

Организация работы с файлами происходит следующим образом:

На примере создания файла in, в котором содержится количество принятых пакетов

1) Создаем функцию для чтения из файла

```
static ssize_t out_show(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf)
{
  unsigned long flags;
  spin_lock_irqsave(&my_lock_1,flags);
```

```
  int a = out_res; int b = out_size_res;
  spin_unlock_irqrestore(&my_lock_1,flags);

  return sprintf(buf, "Packets out: %i; size: %i\n", a,b);
}
```

Листинг 1

2) Выставляем необходимые для него атрибуты

```
static struct kobj_attribute in_attribute =
        __ATTR(in, 0444, in_show, NULL);
```

Листинг 2

где in — имя файла;
0444 — аттрибут «только для чтения»;
in_show — раннее описанная функция для чтения;
NULL говорит о том, что у этого файла нет функции для записи.

3) Создаем список атрибутов для нашего kobject (сюда входят и атрибуты для файла out)

```
static struct attribute *attrs[] = {
   &in_attribute.attr,
   &out_attribute.attr,
   NULL,   /* need to NULL terminate the list of attributes */
 };

 static struct attribute_group attr_group = {
        .attrs = attrs,
 };
```

Листинг 3

4) Объявляем наш kobject

```
static struct kobject *my_kobj
```

5) Создаем простой kobject с именем «my_kobject» в /sys/kernel/

```
my_kobj = kobject_create_and_add("my_kobject", kernel_kobj);
```

6) Создаем файлы, ассоциированные с этим kobject (in и out)

```
retval = sysfs_create_group(my_kobj, &attr_group);
if (retval)
kobject_put(my_kobj)
```

2.4 Описание функций модуля

```
static int mon_notify(struct notifier_block *self, unsigned long action,
                      void *dev)
{
  struct usb_bus *ubus = (struct usb_bus*)dev;
        if (action == USB_BUS_ADD) //если добавлено новое устройство
        {
          ubus->monitored = 1;
          mb();
        }
        return NOTIFY_OK;
}
```

Листинг 5

Данная функция уведомляет о том, что в систему добавлено новое устройство, и устанавливает поле monitored (просмотрено) в 1.

```
static void mon_complete(struct usb_bus *ubus, struct urb *urb, int status)
{
  if (urb->transfer_flags == URB_DIR_IN)
```

```
 {
  spin_lock(&my_lock);
  ++in;
  in_size += urb->actual_length;
  spin_unlock(&my_lock);
 }
  if (urb->transfer_flags == URB_DIR_OUT)
 {
  spin_lock(&my_lock);
  ++out;
  out_size+=urb->actual_length;
  spin_unlock(&my_lock);
 }

}
```

Листинг 6

Эта функция подсчитывает, сколько подтвержденных пакетов. Если у передаваемого пакета флаг направления URB_DIR_IN, то увеличивается переменная in, которая отвечает за количество принятых пакетов, а так же к перемнной in_size добавится размер очередного пакета — urb->actual_length. Если же флаг URB_DIR_OUT, то увеличится переменные out и out_size, которые отвечают за количество и размер переданных пакетов.

```
static int my_print(void * a)
{
 unsigned long flags;

 while(!kthread_should_stop())
 {
  spin_lock_irqsave(&my_lock,flags);
  int a = in; int b = out; int c = in_size; int d = out_size;
  in = 0; out = 0; in_size = 0; out_size = 0;
  spin_unlock_irqrestore(&my_lock,flags);

  spin_lock_irqsave(&my_lock_1,flags);
```

```
    in_res = a; in_size_res = c; out_res = b; out_size_res = d;
    spin_unlock_irqrestore(&my_lock_1,flags);
    pr_notice(" In \n size: %i , number : %i\n", a, c);
    pr_notice(" Out \n size: %i , number : %i\n", b, d);
    msleep(5000);

    }

return 0;
}
```

Листинг 7

Эта функция каждые пять секунд выводит в буфер сообщений ядра информацию о пакетах - количество и размер.

Далее идут функции работы с файлами в sysfs, а также функции загрузки и выгрузки модуля, которые уже были описаны выше.

3. Технологический раздел

Модуль разрабатывался для версии ядра Linux 3.8.6

3.1 Структуры данных

```
static struct notifier_block mon_nb = {
        .notifier_call =        mon_notify,
};
```

Ядро содержит специальный механизм, который называется «notifiers» – «уведомители», который позволяет коду запрашивать, чтобы его оповещали об «интересных» событиях в системе.

Интерфейс notifier выглядит так:

```
struct notifier_block
{
    int (*notifier_call)(struct notifier_block *self,
                unsigned long event, void *data);
    struct notifier_block *next;
    int priority;
};
```

notifier_call() – собственно, и говорит, о каком событии уведомлять. У нас это USB_BUS_ADD

Структура:

```
static struct usb_mon_operations mon_ops_0 = {
        .urb_submit = mon_submit,
        .urb_submit_error = mon_submit_error,
        .urb_complete =        mon_complete,
};
```

Тип:

```
struct usb_mon_operations {
    void (*urb_submit)(struct usb_bus *bus, struct urb *urb);
    void (*urb_submit_error)(struct usb_bus *bus, struct urb *urb, int err);
     void (*urb_complete)(struct usb_bus *bus, struct urb *urb, int status);
};
```

Структура, полями которой являются функции, отвечающие за состояние urb:

urb_submit – пакет сформирован;

urb_submit_error – ошибка при формировании;

ubr_complete — пакет готов и отравлен.

В качестве urb_submit идет функция mon_submit, качестве urb_submit_error идет функция mon_submit_error, a ubr_complete это mon_complete. Так как нас интересуют только пакеты, которые готовы и отправлены, то содержание имеет только фукнция mon_copmlete, остальные же просто объявлены.

3.2 Блокировки в ядре Linux

Библиотека блокировок ядра Linux содержит в себе крайне разнообразные инструменты для обеспечения работы критических секций в коде. Мы рассмотрим несколько инструментов:

Спинлоки

Крайне быстрый метод взаимоблокировки, который сводится к введению ожидающего процессора в бесконечный цикл. Подобный метод взаимоблокировки может привести к входу процессора в бесконечный цикл, если спинлок используется в прерывании или в другом потоке на однопроцессорной машине, поэтому в таких случаях спинлок оптимизируется в пустую операцию. Однако, операции со спинлоками в ядре часто совмещены с операцией отключения прерываний, что само по себе обеспечивает блокировку на однопроцессорной системе. Поэтому если использовать спинлоки с отключением прерываний в обычном коде и

простые спинлоки в коде прерываний, можно достичь очень быстрой работы. Спинлоки подходят для очень коротких критических секций, не более пары строк кода. Спинлоки инициализируются процедурой spinlock_init, занятие и освобождение их выполняют процедуры spin_lock и spin_unlock, а отключающие прерывания версии spin_lock_irqsave и spin_unlock_irqrestore. Спинлоки не требуют освобождения памяти.

Мьютексы

Более продвинутый, но более медленный метод блокировки. Мьютексы позволяют переключать процессы, пока данный поток ждет. Это обеспечивается за счет более сложной архитектуры мьютексов, так что операции занятия на них достаточно трудоемки. К тому же, мьютексы нельзя использовать в прерываниях. Однако они отлично подходят для синхронизации больших и потенциально долгих критических секций. Мьютексы инициализируются процедурой mutex_init, занимаются и освобождаются процедурами mutex_lock и mutex_unlock, уничтожаются процедурой mutex_free.

Завершения

Крайне удобный вариант блокировки для ожидания какого либо события от другого потока. Завершения позволяют одному потоку ожидать (wait_completion) события от другого потока (complete если пребуется пробудить только один поток из очереди, или complete_all если нужно пробудить все). Завершения инициализируются процедурой init_completion и уничтожаются процедурой free_completion.

3.3 Использование потоков ядра

Потоки ядра в ядре Linux проще в использовании чем libpthread или похожие библиотеки потоков в режиме пользователя. Новый поток создается процедурой kthread_create или макросом kthread_run который сразу запускает созданный поток. Процедура потока должна после окончания дождаться вызова kthread_should_exit со стороны создавшего потока, для чего в конце можно создать бесконечный цикл. Со стороны создавшего потока необходимо дождаться завершения работы процедурой kthread_exit.

В нашем модуле создается поток для записи данных о пакетах в буфер сообщений ядра
*struct task_struct *init_thread;*

init_thread = kthread_run(&my_print, NULL, "my_thread");

В модуле спинлоками защищены переменные:
static int in;
static int out;
static int in_size;
static int out_size;
static int in_res;
static int out_res;
static int in_size_res;
static int out_size_res;

Для этого использовалось два спинлока:
static DEFINE_SPINLOCK(my_lock);
static DEFINE_SPINLOCK(my_lock_1);

3.4 Выбор языка и среды программирования

Ядро Linux написано на языке C с небольшим количеством кода на ассемблере, и его система сборки, вообще говоря, поддерживает только данные языки для использования в модулях. Выбор из них был сделан в пользу C по причине значительно большей структурированности написанного на нем кода.

Для сборки драйвера используется сборочная система ядра make. Особенностью ядра Linux является отсутствие совместимости на бинарном уровне, совместимость присутствует лишь на уровне исходных текстов, вследствие чего все модули и само ядро должны быть собраны одной и той же версией одного и того же компилятора.

В качестве среды для написания кода выбран gedit – свободный текстовый редактор рабочей среды GNOME, MAC OC X и Microsoft Windows с поддержкой Юникода. Распространяется на условиях GNU General Public License.

Заключение

Данный проект был направлен прежде всего на изучение процесса написания модулей, работающих в режиме ядра, и эта задача была выполнена. Так же была изучена работа usb-шины, файловая система sysfs. Написан модуль, осуществляющий наблюдение за обменом пакетов между хостом и usb-устройством.

Модуль может быть загружен и выгружен в любой момент, не влияет на работу других устройств в системе, не приводит к задержкам в их работе.

Список используемых источников

1) Исходники ядра Linux и документация к ним

2) http://www.opennet.ru/base/dev/write_linux_driver.txt.html

3) The Linux USB Project: http://www.linux-usb.org/

4) Programming Guide for Linux USB Device Drivers: http://usb.cs.tum.edu/usbdoc

5) Greg Kroan-Hartman. USB Skeleton driver. usb-skeleton.c. 2004

6) Linux Device Drivers, Third Edition by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman 2005

7) http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/samples/kobject/kobject-example.c

8)Олег Цилюрик. Модули ядра Linux.— редакция 4.95 от 10.08.2011

Приложение

1. Руководство пользователя

1. Открыть консоль, перейти в папку, где находится файл *MyMonitor.ko;*

2. Перейти в режим суперпользователя;

3. Набрать в консоли *insmod MyMonitor.ko;*

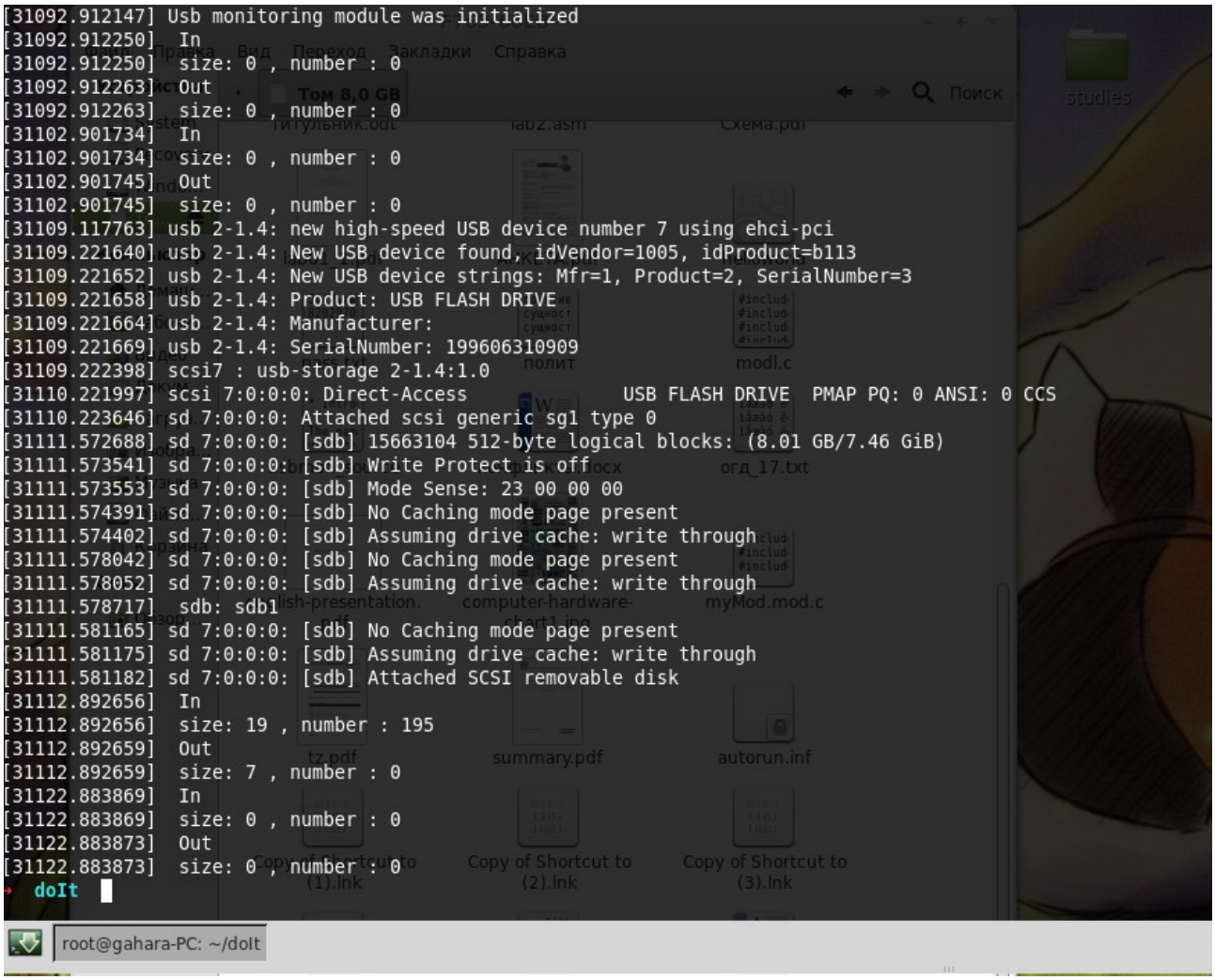
4. Для просмотра информации, выдаваемо модулем, есть два пути:

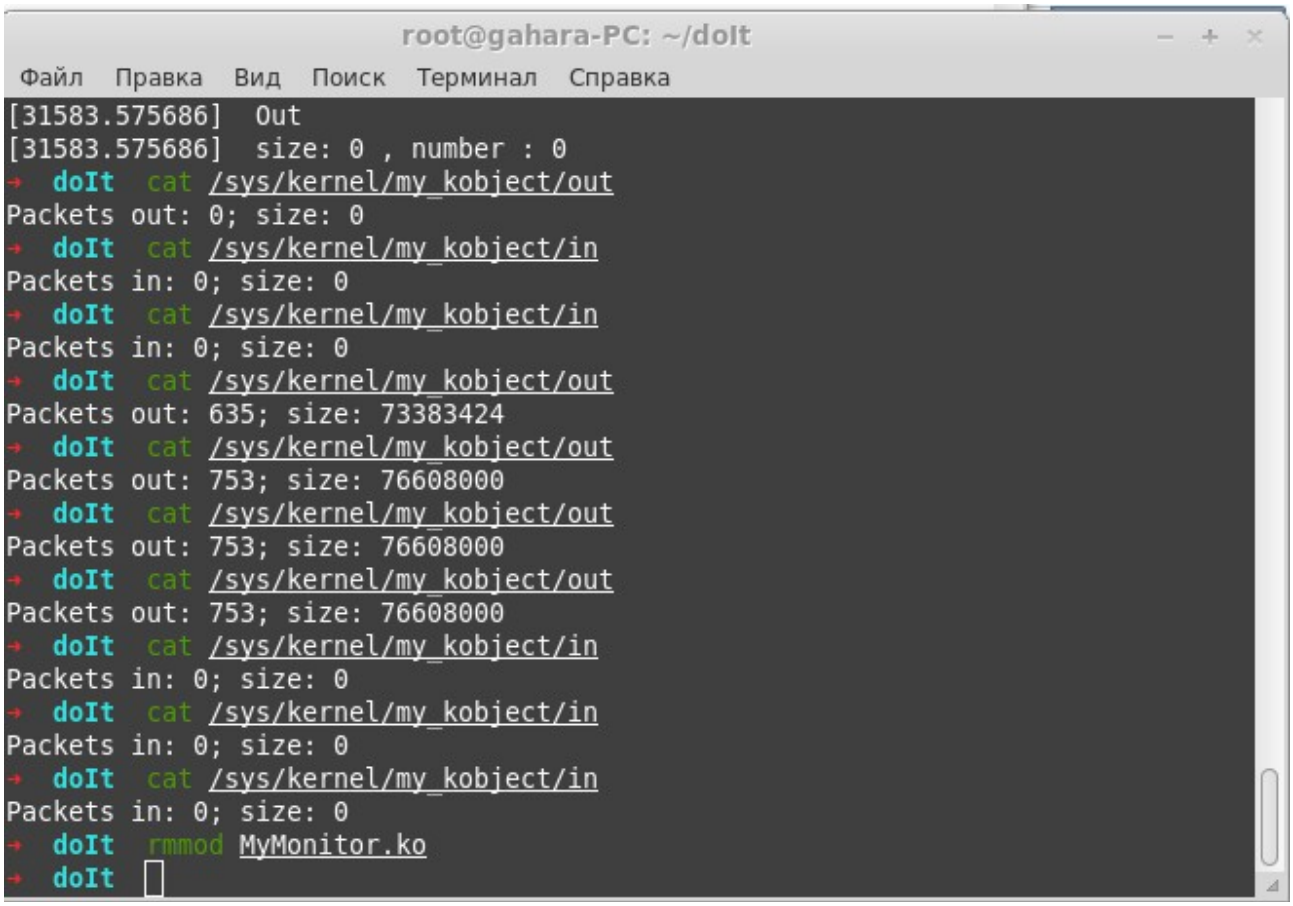
а) Набрать в консоли *dmesg* (команда, используемая в UNIX-подобных операционных системах для вывода буфера сообщений ядра в стандартный поток вывода (stdout) (по умолчанию на экран);

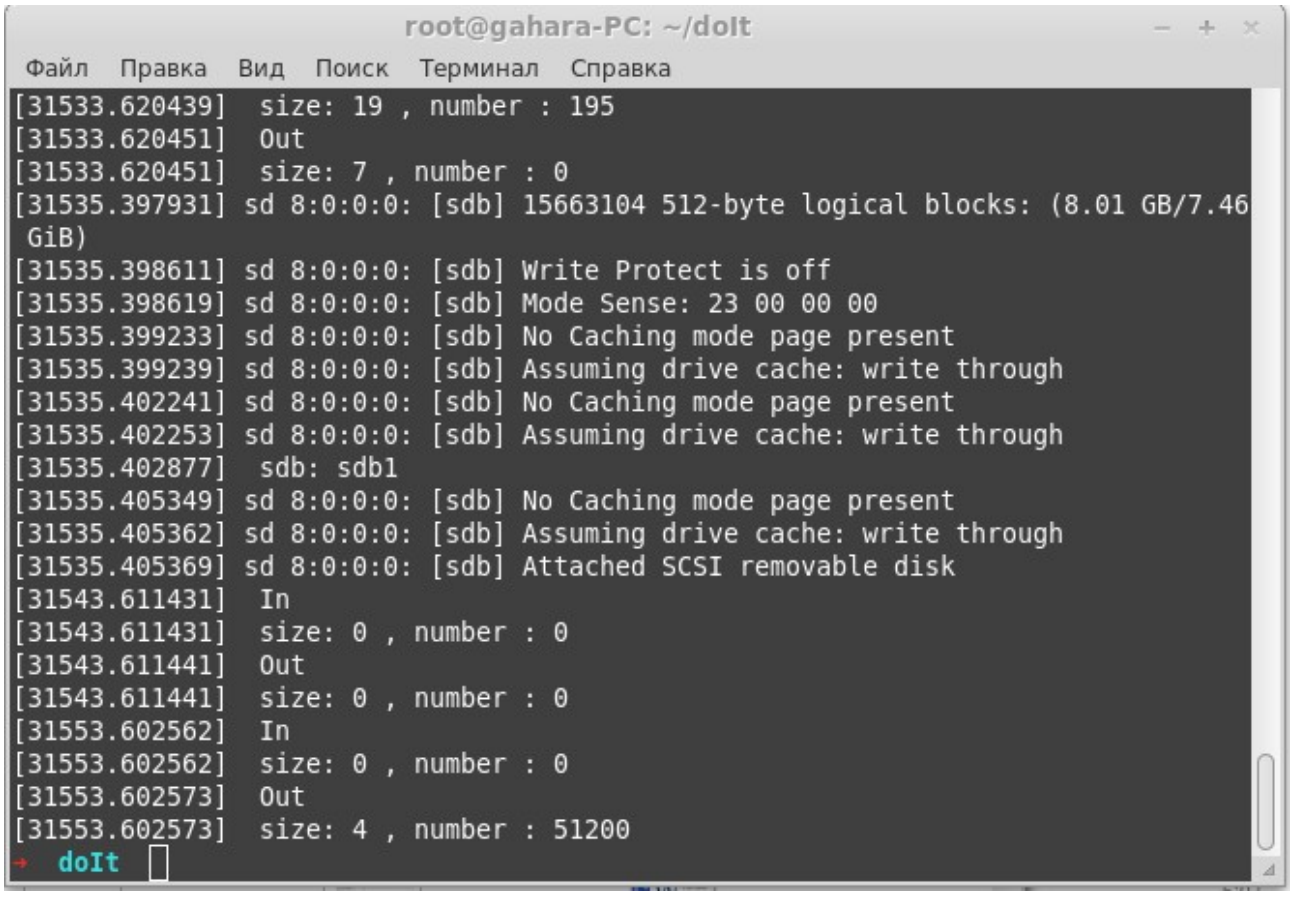
б) Набрать в консоли *cat /sys/kernel/my_kobject/in* или *cat /sys/kernel/my_kobject/out;*

5. Для выгрузки модуля сделать *rmmod MyMonitor.ko*

2. Примеры работы модуля




root@gahara-PC: ~/doIt
Файл Правка Вид Поиск Терминал Справка
[31583.575686] Out
[31583.575686] size: 0 , number : 0
→ doIt cat /sys/kernel/my_kobject/out
Packets out: 0; size: 0
→ doIt cat /sys/kernel/my_kobject/in
Packets in: 0; size: 0
→ doIt cat /sys/kernel/my_kobject/in
Packets in: 0; size: 0
→ doIt cat /sys/kernel/my_kobject/out
Packets out: 635; size: 73383424
→ doIt cat /sys/kernel/my_kobject/out
Packets out: 753; size: 76608000
→ doIt cat /sys/kernel/my_kobject/out
Packets out: 753; size: 76608000
→ doIt cat /sys/kernel/my_kobject/out
Packets out: 753; size: 76608000
→ doIt cat /sys/kernel/my_kobject/in
Packets in: 0; size: 0
→ doIt cat /sys/kernel/my_kobject/in
Packets in: 0; size: 0
→ doIt cat /sys/kernel/my_kobject/in
Packets in: 0; size: 0
→ doIt rmmod MyMonitor.ko
→ doIt


root@gahara-PC: ~/doIt
Файл Правка Вид Поиск Терминал Справка
[31533.620439] size: 19 , number : 195
[31533.620451] Out
[31533.620451] size: 7 , number : 0
[31535.397931] sd 8:0:0:0: [sdb] 15663104 512-byte logical blocks: (8.01 GB/7.46
GiB)
[31535.398611] sd 8:0:0:0: [sdb] Write Protect is off
[31535.398619] sd 8:0:0:0: [sdb] Mode Sense: 23 00 00 00
[31535.399233] sd 8:0:0:0: [sdb] No Caching mode page present
[31535.399239] sd 8:0:0:0: [sdb] Assuming drive cache: write through
[31535.402241] sd 8:0:0:0: [sdb] No Caching mode page present
[31535.402253] sd 8:0:0:0: [sdb] Assuming drive cache: write through
[31535.402877] sdb: sdb1
[31535.405349] sd 8:0:0:0: [sdb] No Caching mode page present
[31535.405362] sd 8:0:0:0: [sdb] Assuming drive cache: write through
[31535.405369] sd 8:0:0:0: [sdb] Attached SCSI removable disk
[31543.611431] In
[31543.611431] size: 0 , number : 0
[31543.611441] Out
[31543.611441] size: 0 , number : 0
[31553.602562] In
[31553.602562] size: 0 , number : 0
[31553.602573] Out
[31553.602573] size: 4 , number : 51200
→ doIt