Санкт-Петербургский Политехнический университет

Институт Компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Лабораторная работа №4

Дисциплина: Проектирование ОС и компонентов

Тема: Встраивание модулей в ядро.

Разработка драйвера (на примере символьного устройства)

Выполнил студент гр. 13541/4 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. И. Шайтан

(подпись)

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е. В. Душутина

(подпись)

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2017 г.

Санкт-Петербург

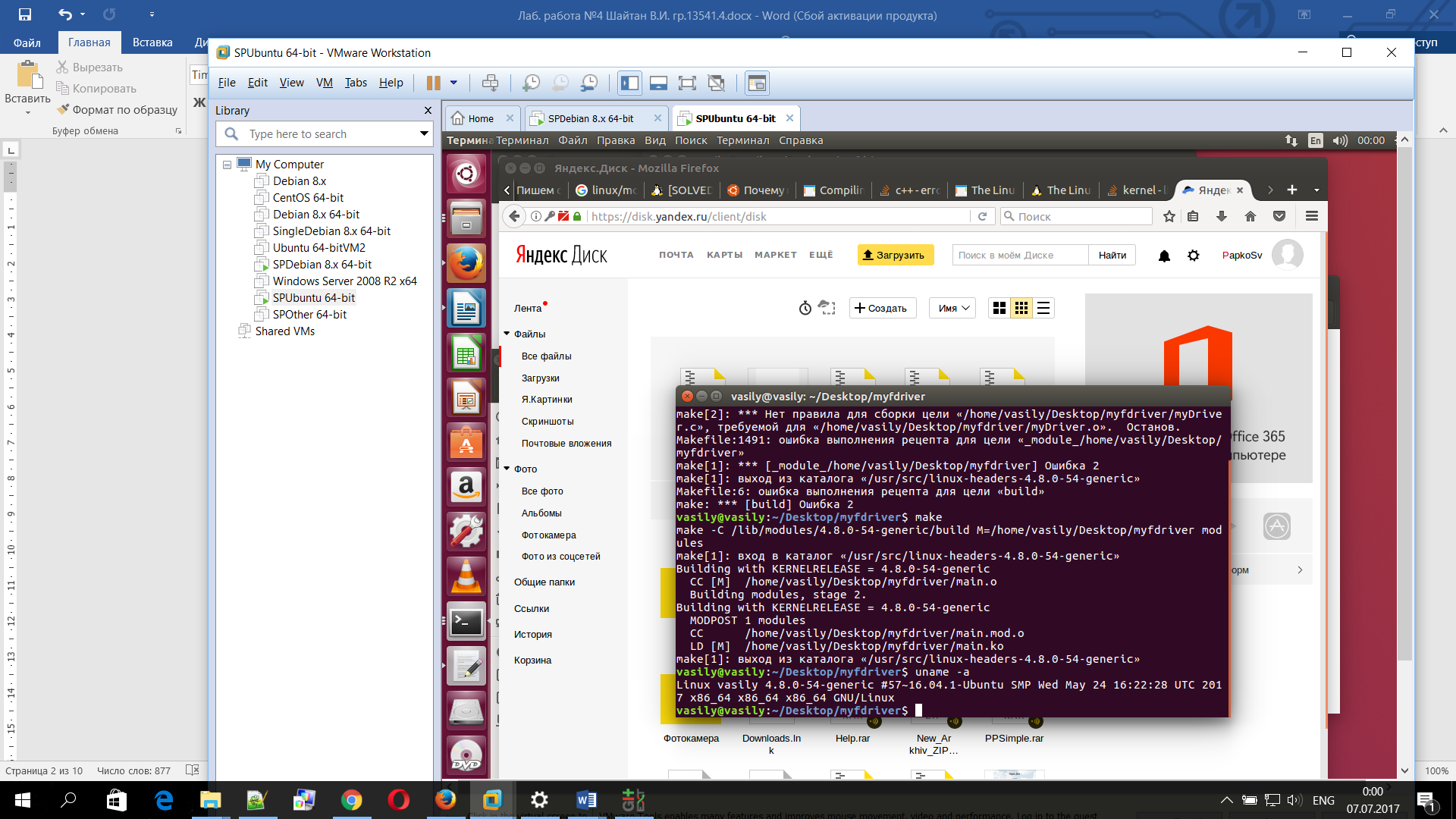
2017

**Выполнение работы:**

**Цели работы:**

В данной работе необходимо изучить способы создания драйвера символьного устройства и создать собственный драйвер в ОС Linux.

В данной работе, для написания драйвера символьного устройства была использована ОС Ubuntu 16.04, установленная на виртуальную машину. Сделано это было во избежание нанесения непоправимого ущерба рабочей системе. Информация об использованной системе приведена ниже:



Драйвер символьного устройства – драйвер для байт-ориентированных операций. Для любого приложения, предназначенного для работы с байт-ориентированным устройством, следует использовать соответствующий драйвер символьного устройства. Использование символьных драйверов осуществляется через соответствующие файлы символьных устройств.

Файлы устройств подключаются к драйверу устройства с помощью специального механизма регистрации, что осуществляется драйвером. Драйвер связывается с устройством с помощью специальных низкоуровневых операций, характерных для конкретного устройства. Таким образом, формируется полное соединение. При этом, файл символьного устройства не является реальным устройством, это просто специальная методика (place-holder) подключения реального устройства.

При подключении приложения к файлу устройства используется имя файла устройства. Но при подключении файла устройства к драйверу устройства используется номер файла устройства, а не имя файла. В результате, приложение пользовательского пространства может использовать для файла устройства любое имя, а в пространстве ядра для связи между файлом устройства и драйвером устройства можно использовать тривиальный механизм индексации. Таким номером файла обычно является пара <major, minor>, т. е. или старший и младший номера файла устройства.

Ранее (вплоть до ядра 2.4) каждый старший номер использовался в качестве указания на отдельный драйвер, а младший номер использовался для указания на конкретное подмножество функциональных возможностей драйвера. В ядре 2.6 такое использование номеров не является обязательным; с одним и тем же старшим номером может быть несколько драйверов, но, очевидно, с различными диапазонами младших номеров.

Однако, такое использование больше характерно для незарезервированных старших номеров, а стандартные старшие номера обычно резервируются для вполне определенных конкретных драйверов. Например, 4 — для последовательных интерфейсов, 13 - для мышей, 14 — для аудиоустройств и так далее.

Для выполнения данной работы был написан простой модуль ядра, работающий с файлом устройства /dev/test

Полный код представлен чуть ниже.

Рассмотрим код подробнее.

Первое, что необходимо сделать – указать необходимые заголовочные файлы:

|  |
| --- |
| #include <linux/kernel.h> /\* Для printk() и т.д. \*/  #include <linux/module.h> /\* Оживляет модули \*/  #include <linux/init.h> /\* Определения макросов \*/  #include <linux/fs.h>  #include <asm/uaccess.h> /\* put\_user \*/ |

Первая строчка подключает linux/kernel.h. Этот заголовочный файл позволяет использовать функцию printk(). Данная функция ядра работает примерно так же, как и функция printf(), только имеет форматирование специально для ядра. Основное назначение этой функции — реализация механизма регистрации событий и предупреждений. Иными словами, эта функция для записи в лог ядра некой информации.

Заголовочные файлы linux/module.h и linux/init.h требуются для всех модулей. Они включают в себя различные определения макросов, таких как module\_init(), module\_exit() и MODULE\_\*().

Заголовочный файл linux/fs.h содержит в себе указатели на функции. Каждое поле структуры соответствует адресу некоторой функции, определенной драйвером для обработки запрошенной операции.

Файл asm/uaccess.h обеспечивает доступ к функции put\_user(). Данная функция необходима, т.к. драйвер работает в пространстве ядра, то он ограничен от адресного пространства пользователя. И для того, чтобы вернуть какой-нибудь результат используется функция put\_user().

Следующее, что хотелось бы рассмотреть – информацию о модуле:

|  |
| --- |
| MODULE\_LICENSE( "GPL" );  MODULE\_AUTHOR( "Vasily Shaytan <v.shaytan@gmail.com>" );  MODULE\_DESCRIPTION( "My first module" );  MODULE\_SUPPORTED\_DEVICE( "test" ); /\* /dev/testdevice \*/ |

Здесь можно указать лицензию, под которой будет распространяться модуль ядра, имя автора, описание модуля и поддерживаемое устройство.

Далее указываем операции, поддерживаемые устройством и их обработчики:

|  |
| --- |
| // Поддерживаемые нашим устройством операции  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 \* );  // Глобальные переменные, объявлены как static, воизбежание конфликтов имен.  static int major\_number; /\* Старший номер устройства нашего драйвера \*/  static int is\_device\_open = 0; /\* Используется ли девайс ? \*/  static char text[ 5 ] = "test\n"; /\* Текст, который мы будет отдавать при обращении к нашему устройству \*/  static char\* text\_ptr = text; /\* Указатель на текущую позицию в тексте \*/  // Прописываем обработчики операций на устройством  static struct file\_operations fops =  {  .read = device\_read,  .write = device\_write,  .open = device\_open,  .release = device\_release  }; |

Здесь указываются возможные операции чтения, записи, открытия и закрытия.

Далее создаются точки входа и выхода из модуля. Определяются макросы \_\_init, \_\_exit.

|  |
| --- |
| // Функция загрузки модуля. Входная точка. Можем считать что это наш main()  static int \_\_init test\_init( void )  {  printk( KERN\_ALERT "TEST driver loaded!\n" );  // Регистрируем устройсво и получаем старший номер устройства  major\_number = register\_chrdev( 0, DEVICE\_NAME, &fops );  if ( major\_number < 0 )  {  printk( "Registering the character device failed with %d\n", major\_number );  return major\_number;  }  // Сообщаем присвоенный нам старший номер устройства  printk( "Test module is loaded!\n" );  printk( "Please, create a dev file with 'mknod /dev/test c %d 0'.\n", major\_number );  return SUCCESS;  }  // Функция выгрузки модуля  static void \_\_exit test\_exit( void )  {  // Освобождаем устройство  unregister\_chrdev( major\_number, DEVICE\_NAME );  printk( KERN\_ALERT "Test module is unloaded!\n" );  }  // Указываем наши функции загрузки и выгрузки  module\_init( test\_init );  module\_exit( test\_exit ); |

В функции загрузки производится регистрация устройства и присвоение ему пары номеров (старшего и младшего). Сообщение пользователю присвоенных старшего и младшего номеров.

В функции выгрузки происходит только освобождение устройства.

Далее производим реализацию методов, открытия, закрытия, чтения и записи в файл.

|  |
| --- |
| static int device\_open( struct inode \*inode, struct file \*file )  {  text\_ptr = text;  if ( is\_device\_open )  return -EBUSY;  is\_device\_open++;  return SUCCESS;  }  static int device\_release( struct inode \*inode, struct file \*file )  {  is\_device\_open--;  return SUCCESS;  } |

В методе открытия устройства задаем отображаемый текст, и меняем флаг открытого устройства. В методе закрытия – устанавливается флаг открытого устройства в 0.

Далее – реализация методов чтения и записи:

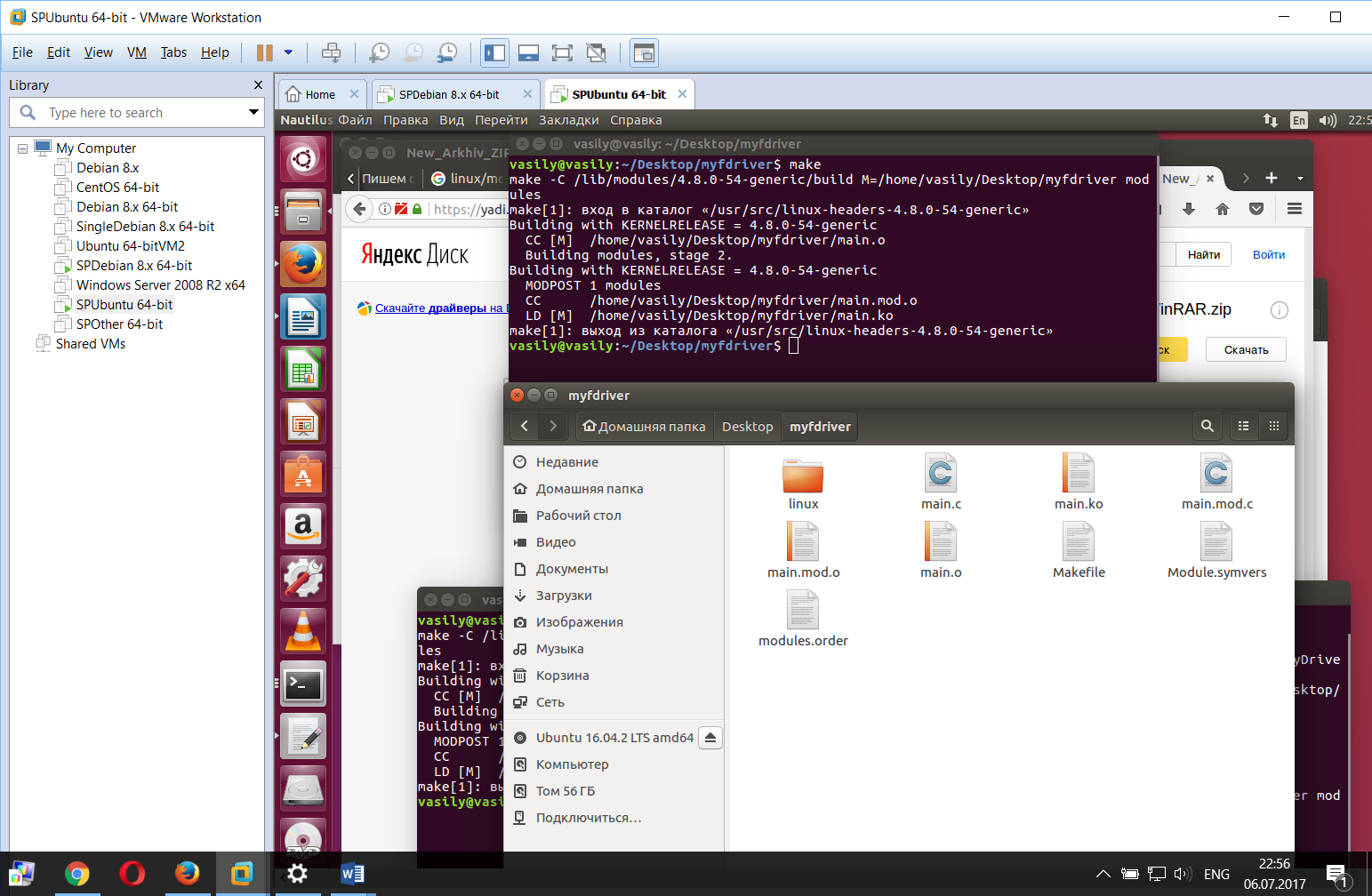
|  |
| --- |
| static ssize\_t  device\_write( struct file \*filp, const char \*buff, size\_t len, loff\_t \* off )  {  printk( "Sorry, this operation isn't supported.\n" );  return -EINVAL;  }  static ssize\_t device\_read( struct file \*filp, /\* include/linux/fs.h \*/  char \*buffer, /\* buffer \*/  size\_t length, /\* buffer length \*/  loff\_t \* offset )  {  int byte\_read = 0;  if ( \*text\_ptr == 0 )  return 0;  while ( length && \*text\_ptr )  {  put\_user( \*( text\_ptr++ ), buffer++ );  length--;  byte\_read++;  }  return byte\_read;  } |

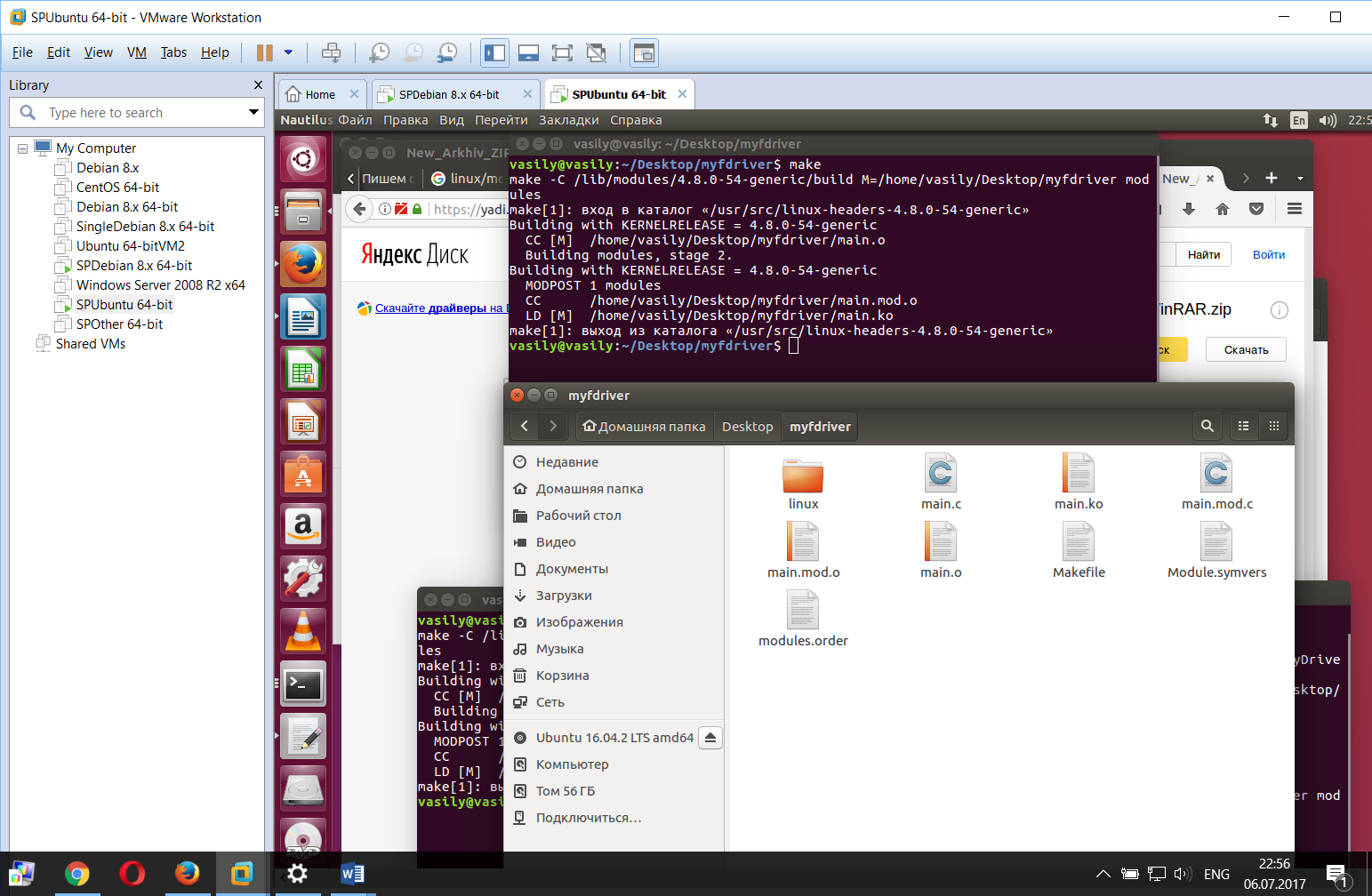
В методе записи в файл будет только уведомление о неподдерживаемой операции. А в методе чтения – считывание всей информации и запись в dmesg сообщения о том, что данный метод был вызван.

Теперь, после написания кода модуля ядра, его необходимо собрать. Для этого используется Makefile следующего содержания:

|  |
| --- |
| ifeq ($(KERNELRELEASE),)  KERNELDIR ?= /lib/modules/$(shell uname -r)/build  PWD := $(shell pwd)  .PHONY: build clean  build:  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules  clean:  rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c  else  $(info Building with KERNELRELEASE = ${KERNELRELEASE})  obj-m := main.o  endif |

Теперь запускаем его командой make:





Для работы с модулем понадобятся следующие команды:

• insmod – добавление модуля в ядро

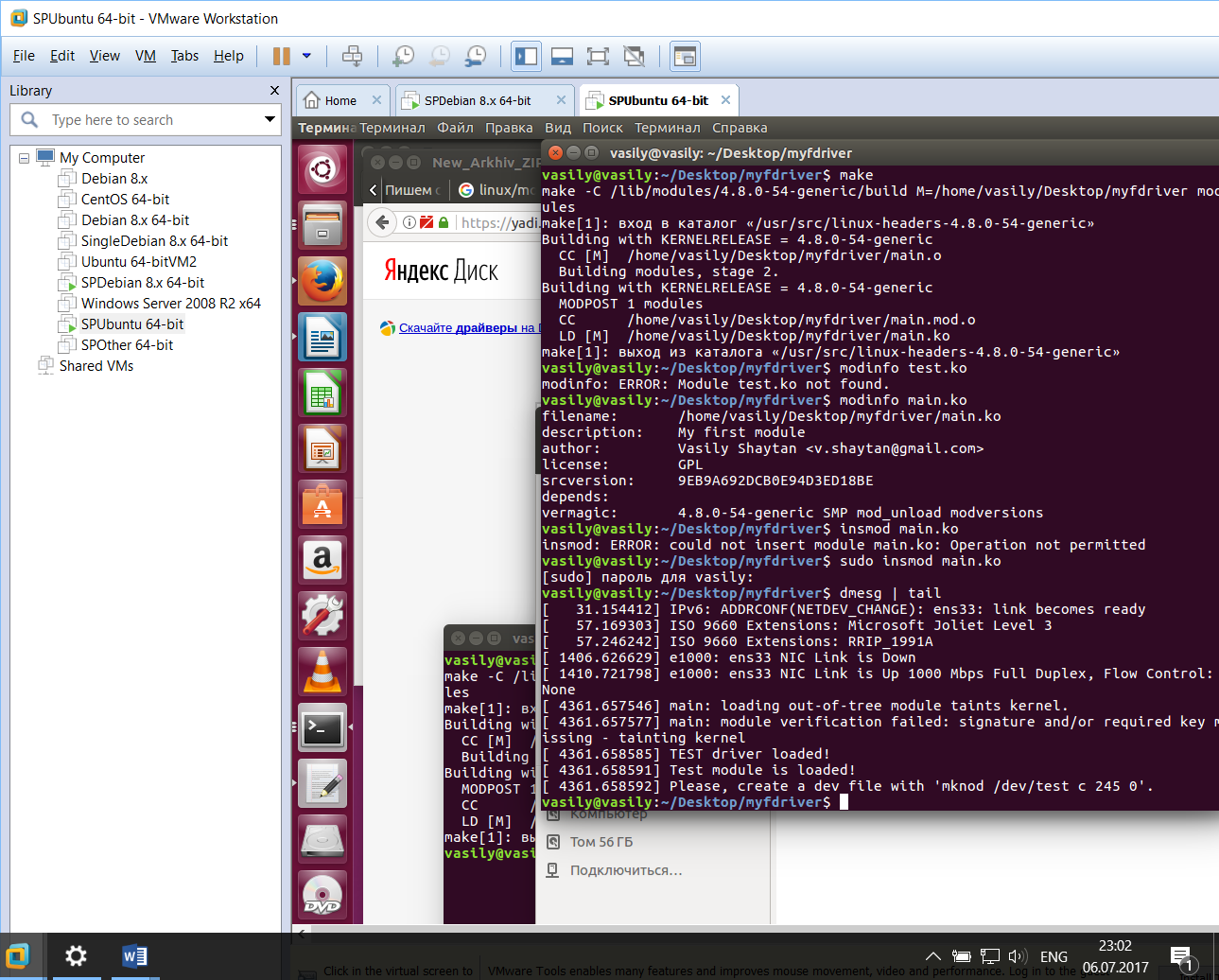
• rmmod – удаление модуля из ядра

• lsmod – вывод списка текущих модулей

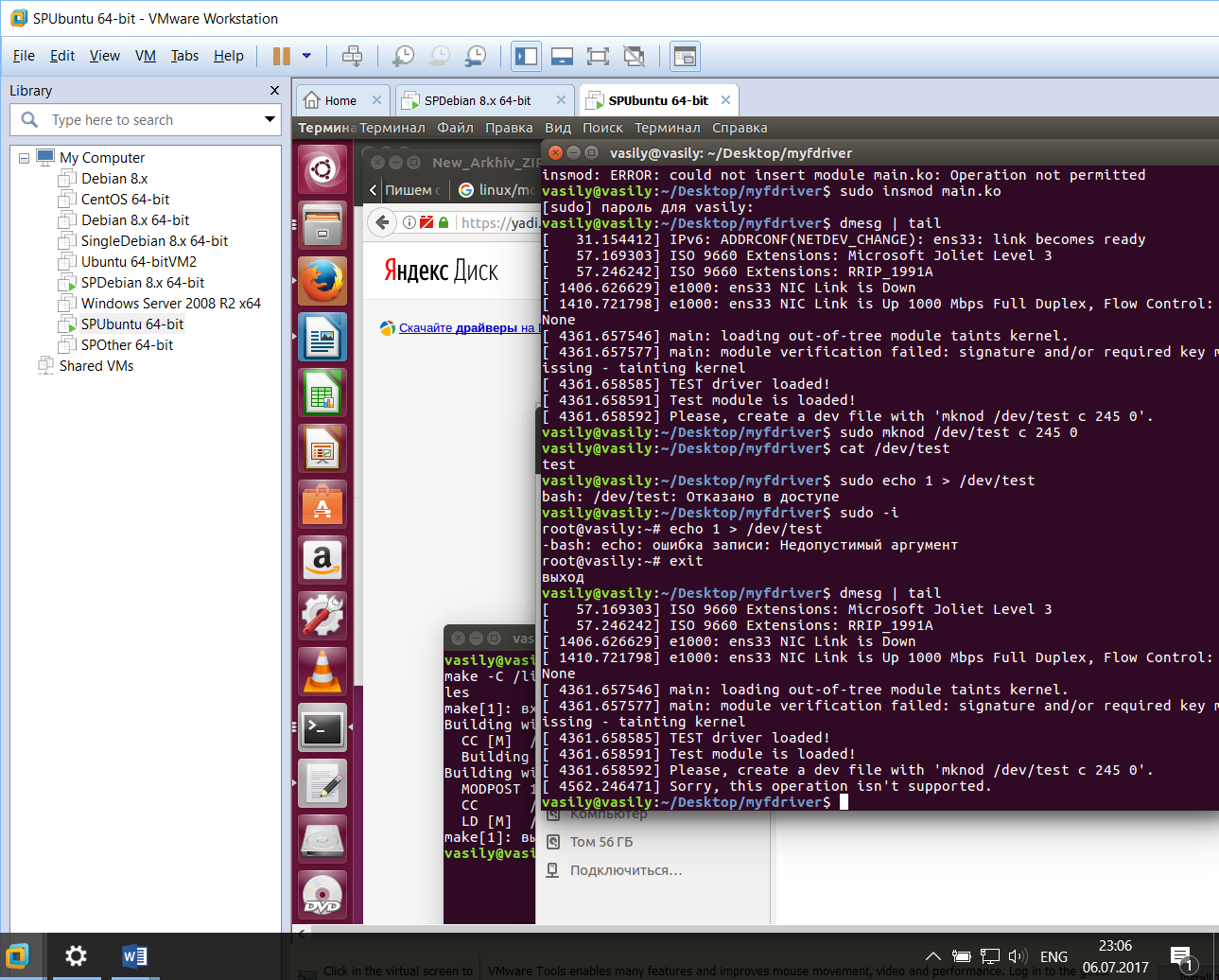
• modinfo – вывод информации о модуле

Для начала, посмотрим информацию о полученном модуле.

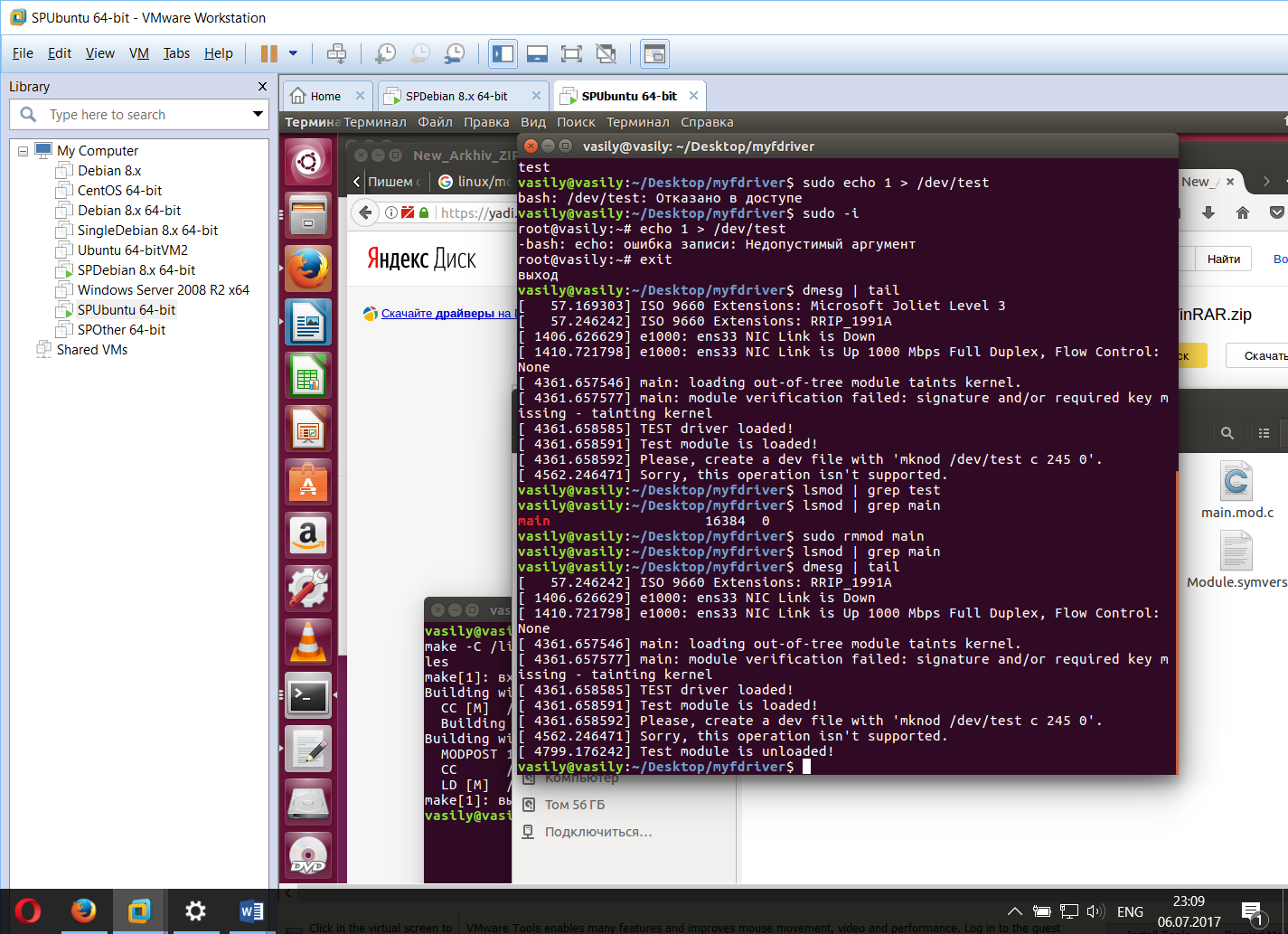
Из полученной информации можно увидеть, что указана именно та информация, которая была указана в коде. Далее произведем вставку модуля в ядро и проверим его работу.



Так как в системе отсутствует файл /dev/test, создадим его с помощью команды mknod и протестируем методы чтения и записи:



После того, как была произведена проверка работоспособности написанного модуля, его можно удалить из ядра:



Полный код драйвера:

|  |
| --- |
| #include <linux/kernel.h> /\* Для printk() и т.д. \*/  #include <linux/module.h> /\* Оживляет модули \*/  #include <linux/init.h> /\* Определения макросов \*/  #include <linux/fs.h>  #include <asm/uaccess.h> /\* put\_user \*/  // Ниже мы задаём информацию о модуле, которую можно будет увидеть с помощью Modinfo  MODULE\_LICENSE( "GPL" );  MODULE\_AUTHOR( "Vasily Shaytan <v.shaytan@gmail.com>" );  MODULE\_DESCRIPTION( "My first module" );  MODULE\_SUPPORTED\_DEVICE( "test" ); /\* /dev/testdevice \*/  #define SUCCESS 0  #define DEVICE\_NAME "test" /\* Имя нашего устройства \*/  // Поддерживаемые нашим устройством операции  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 \* );  // Глобальные переменные, объявлены как static, воизбежание конфликтов имен.  static int major\_number; /\* Старший номер устройства нашего драйвера \*/  static int is\_device\_open = 0; /\* Используется ли девайс ? \*/  static char text[ 5 ] = "test\n"; /\* Текст, который мы будет отдавать при обращении к нашему устройству \*/  static char\* text\_ptr = text; /\* Указатель на текущую позицию в тексте \*/  // Прописываем обработчики операций на устройством  static struct file\_operations fops =  {  .read = device\_read,  .write = device\_write,  .open = device\_open,  .release = device\_release  };  // Функция загрузки модуля. Входная точка. Можем считать что это наш main()  static int \_\_init test\_init( void )  {  printk( KERN\_ALERT "TEST driver loaded!\n" );  // Регистрируем устройсво и получаем старший номер устройства  major\_number = register\_chrdev( 0, DEVICE\_NAME, &fops );  if ( major\_number < 0 )  {  printk( "Registering the character device failed with %d\n", major\_number );  return major\_number;  }  // Сообщаем присвоенный нам старший номер устройства  printk( "Test module is loaded!\n" );  printk( "Please, create a dev file with 'mknod /dev/test c %d 0'.\n", major\_number );  return SUCCESS;  }  // Функция выгрузки модуля  static void \_\_exit test\_exit( void )  {  // Освобождаем устройство  unregister\_chrdev( major\_number, DEVICE\_NAME );  printk( KERN\_ALERT "Test module is unloaded!\n" );  }  // Указываем наши функции загрузки и выгрузки  module\_init( test\_init );  module\_exit( test\_exit );  static int device\_open( struct inode \*inode, struct file \*file )  {  text\_ptr = text;  if ( is\_device\_open )  return -EBUSY;  is\_device\_open++;  return SUCCESS;  }  static int device\_release( struct inode \*inode, struct file \*file )  {  is\_device\_open--;  return SUCCESS;  }  static ssize\_t  device\_write( struct file \*filp, const char \*buff, size\_t len, loff\_t \* off )  {  printk( "Sorry, this operation isn't supported.\n" );  return -EINVAL;  }  static ssize\_t device\_read( struct file \*filp, /\* include/linux/fs.h \*/  char \*buffer, /\* buffer \*/  size\_t length, /\* buffer length \*/  loff\_t \* offset )  {  int byte\_read = 0;  if ( \*text\_ptr == 0 )  return 0;  while ( length && \*text\_ptr )  {  put\_user( \*( text\_ptr++ ), buffer++ );  length--;  byte\_read++;  }  return byte\_read;  } |

**Вывод**

В ходе данной лабораторной работы были изучены методы создания и запуска модулей ядра (драйверов символьного устройства).

Помимо изучения теоретической части, была произведена практическая реализация. То есть, был создан простой модуль ядра, взаимодействующий с файлом устройства /dev/test.

**Список используемых источников:**

1. <https://habrahabr.ru/post/106702/> - Пишем свой драйвер под Linux
2. <http://www.opennet.ru/base/dev/dev_linux_modules.txt.html> - Введение в написание модулей ядра Linux. (linux kernel module gcc)