МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ
Зав. кафедрой АСУ, профессор

А.М. Кориков

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Тема 3. Udev и псевдофайловые системы

Учебно-методическое пособие

для студентов уровня основной образовательной программы: **магистратура**
направление подготовки: **09.04.01 - Информатика и вычислительная техника**

Разработчик
доцент кафедры АСУ

В.Г. Резник

2016

Резник В.Г.

Современные операционные системы. Тема 3. Udev и псевдофайловые системы. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 38 с.

Учебно-методическое пособие предназначено для изучения темы №3 по дисциплине «Современные операционные системы» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Оглавление

Введение

В теме 3 рассматриваются вопросы взаимодействия ядра ОС с аппаратной частью ЭВМ, которое для пользователя ЭВМ выглядит как работа с файлами псевдофайловых систем: *devtmpfs*, *procfs* и *sysfs*, которые в современных операционных системах обслуживаются программным обеспечением проекта *udev*.

В теоретическом материале рассматриваются следующие вопросы:

- Проблематика работы с устройствами ЭВМ.
- Назначение и функции udev (devfs, hotplug и HAL).
- Псевдофайловая система proc.
- Псевдофайловая система sysfs.

Особое внимание уделяется синтаксису и семантике языка динамического управления устройствами, с помощью которого не только описываются сами устройства, но управление ими с помощью сигналов, отражающих изменение их состояния.

Теоретические знания закрепляются во время проведения лабораторной работы, которая обеспечивает как закрепление теоретических знаний, так и нужные навыки для освоения объявленной темы. По завершению лабораторной работы студент должен уметь самостоятельно реализовывать простейшие приемы динамического управления устройствами, а также эффективно использовать возможности современных псевдофайловых систем.

Методика проведения лабораторных работ по данной дисциплине предполагает использование специального дистрибутива ОС УПК АСУ, которое установлено в компьютерных классах кафедры АСУ.

1 Тема 3. Udev и псевдофайловые системы

В компьютерной индустрии, *понятие устройства* нагружено множеством семантических представлений:

- *для пользователя ЭВМ* — это может быть или сам компьютер или его отдельные компоненты, например, *клавиатура, мышь, жесткий диск*;
- *для пользователя ОС* — это диски *c:, …, z:* или, например, имена файлов в директории */dev*;
- *для системного администратора* — это *набор имен*, отображаемый в различных файлах и директориях файловой системы ОС, которые появляются или удаляются, при установке драйверов устройств, или сообщаются ему системными программами (утилитами); такими именами могут быть, например, *устройство /dev/sda1* — соответствующее файловой системе в первом разделе первого жесткого диска, или *устройство /dev/tty1* — соответствующее первому виртуальному терминалу ОС.

В предыдущей теме было показано, что такие имена возникают при загрузке драйверов, которые посредством функции *init_module()*, в момент инициализации модуля, вызывают также внутренние функции API ядра, регистрирующие имя драйвера и обеспечивая *базовое имя* для отображения имени устройства в директории */dev*.

С другой стороны, было отмечено (см. Тема 2, подраздел 1.8, стр. 32), что любой драйвер должен обеспечивать работу трех интерфейсов:

- *внешний интерфейс* в пространство пользователя;
- *внешний интерфейс* с аппаратными средствами ЭВМ;
- *внутренний интерфейс* к API ядра.

Именно наличие внешнего интерфейса с аппаратными средствами ЭВМ, требует дополнительной семантики для понятия устройства, в целях комплексной реализации системного программного обеспечения ОС.

Таким образом, в дальнейшем будет использоваться понятие — *системное устройство*, которое в зависимости от контекста будет различаться на:

- *hard device* — *аппаратное устройство*, когда мы говорим об аппаратных средствах ЭВМ;
- *soft device* — *программное устройство*, когда мы говорим об именах, отображаемых в системе драйверами ОС.

В целом, тематика данной части учебного курса посвящена проблемам использования системных устройств в современных ОС, которые методически будут раскрыты следующим образом:

- *сначала*, рассмотрим *исторический экскурс* описания этих проблем;
- *затем*, изучим сорременное решение, реализованное в проекте *udev*;
- *наконец*, рассмотрим проекции решения *udev* на другие подсистемы ОС.

1.1 Проблематика работы с устройствами ЭВМ

Превоначально, работа с системными устройствами не вызывала особых затруднений:

- *устройств было мало* и они были хорошо известны как разработчикам ОС, так и администраторам ЭВМ;
- *драйвера устройств писались разработчиками ОС* и, обычно, включались в состав ядра (супервизора);
- *в такой ситуации*, *понятие системного устройства* было практически полностью тождественно именам, которые драйвера устройств отображали в пространство пользователя.

В процессе бурного развития аппаратных средств ЭВМ, стало возникать множество проблем разработки эффективного и надежного системного ПО ОС:

- проблема *увеличения номинального состава оборудования*, для которого необходимо реализовывать различные драйвера;
- проблема *распределения ограниченных ресурсов ЭВМ*, при подключении одинаковых по функциональным возможностям аппаратных средств, но требующих для себя индивидульных ресурсов;
- проблема появления *динамически подключаемых устройств*, которые появляются и конфигурируются в системе не в момент включения ЭВМ, а в процессе активного функционирования ОС;
- проблема *раздельной разработки ОС и загружаемых драйверов*, реализуемых часто разработчиками оборудования.

В результате:

- возникла острая необходимость разделения семантики системного устройства на два понятия, выделив отдельно *hard device* и *soft device*;
- появились системные технологии, называемые *Plag-and-Play, HotPlug, HAL* и *devfs*.

Рассмотрим кратко перечисленные понятия.

1.1.1 Plag-and-Play и HotPlug

Plug and Play (***PnP***), дословно переводится как «*включил и играй (работай)*» — технология, предназначенная для быстрого определения и конфигурирования устройств в компьютере и других технических устройствах.
В плане названия, - этому термину больше доходит понятие *hotplug*.

HotPlug — *горячее подключение* — термин означающий *подключение* электронного оборудования к (компьютерной) системе, во время ее работы без выключения питания и остановки самой системы.
Семантика понятия ***HotPlug*** влючает в себя семантику технологии ***PnP***.

1.1.2 HAL

Hardware Abstraction Layer (***HAL***, *Слой аппаратных абстракций*) — слой абстрагирования, реализованный в системном программном обеспечении, находящийся между физическим уровнем аппаратных средств ЭВМ и программным обеспечением, запускаемом на этом компьютере.
HAL предназначен для сокрытия различий в аппаратном обеспечении от основной части ядра ОС, таким образом, чтобы, при запуске на системах с различным аппаратным обеспечением, большая часть кода ядра не нуждалась в своем изменении.

На персональных компьютерах HAL может рассматриваться как драйвер материнской платы, позволяющий взаимодействовать с низкоуровневыми компонентами, такими, как аппаратное обеспечение, с помощью инструкций высокоуровневых языков программирования.

В операционных системах семейства MS Windows, HAL является неотъемлемой частью кода, исполняемого в режиме ядра, хотя находится в отдельном загрузочном модуле, загружаемом совместно с ядром.

В операционных системах UNIX/Linux, поддержка HAL прекратилась ***в ноябре 2009 года***.

1.1.3 devfs

С начала 2000-х годов, в технологиях ОС, стала применяться файловая система *devfs*, цель проекта которой была решить основную проблему ОС UNIX/Linux, связанную как с катастрофически возрастающим объемом директории */dev*, так и с поддержкой технологии ***PnP***.
Данный проект предполагал использование специального демона *devfsd*, который периодически обращался к ядру ОС с целью контроля за состоянием имеющегося оборудования:

- при обнаружении подключения нового оборудования, этот демон создавал соответствующий узел (***inode***) в директории */dev*;
- в случае обнаружения ситуации, что оборудование отключилось, он соответственно удалял узел из директории */dev*.

Существенным недостатком *devfs* является *неэффективное расходование ресурсов ЭВМ*, связанное с его постоянным циклическим запуском, поэтому современные системы ОС UNIX/Linux перешли на более новую технологию, известную как *udev*.

1.2 Назначение и функции udev

В ОС Linux, технология *udev* появилась ***в ноябре 2003 года***.
Старая технология *devfs*, кроме указанного недостатка — постоянной циклической проверки демоном *devfsd* состояния всех устройств ЭВМ, имела еще один недостаток: "*размножение*" устройств при многократном подключении одного и того же USB-драйвера.

Новая технология *udev* была реализована на идее *обработки событий*.
Теперь драйвер, обнаружив "свое" устройство, запрашивает закрепленные за ним ресурсы и, как правило, "соглашается" с таким распределением. Драйвер, используя функции ядра, может попытаться переопределить ресурсы устройства. В этом случае за отсутствие возможных конфликтов отвечает ядро - *оно просто не принимает адреса, конфликтующие с другими устройствами*.

Программное обеспечение технологии *udev* разделено на три части:

- библиотека *libudev*, позволяющая получать доступ к информации об устройствах;
- демон *udevd*, работающий в пространстве пользователя и управляющий содержимым директории */dev*;
- программа *udevadm*, используемая для отладки и диагностики, а также в качестве интерактивного менеджера для управления устройствами в режиме командной строки терминала.

В апреле 2012 года в ОС Linux, исходный код проекта *udev* слился с исходным кодом проекта *systemd*, в пределах которого он и развивается по настоящее время.
Необходимость такого решения вызвана не наличием недостатков в проекте ***udev***, а потребностями проекта ***systemd***, которому требуется тесное взаимодействие с подсистемой ОС, контролирующей оборудование ЭВМ.
Теперь, функции демона *udevd* выполняет модуль *systemd-udevd*, входящий в новый проект, но имеющий достаточно самостоятельные полномочия.
В частности, именно модуль *systemd-udevd* запускается при старте ОС, обнаруживая и создавая устройства ЭВМ, тем самым, обеспечивая условия подключения корневой файловой системы.

Что касается непосредственно возможностей проекта *systemd*, то этому вопросу посвящена следующая тема нашего курса.
В данной теме, мы рассмотрим основные технологические возможности проекта *udev*.

Современная технология udev, применяемая в ОС Linux, использует не только возможности псевдовайловой системы *procfs* (***proc***), основы которой появились в ОС UNIX еще ***с середины 1980-х годов***, но и:

- ***kobjects*** — новые объекты подсистемы обслуживания устройств;
- ***sysfs*** - новую виртуальную файловую систему, экспортирующую в простран-

ство пользователя информацию ядра о присутствующих в системе устройствах и драйверах.

Автором **sysfs** является *Патрик Мошель* (Patrick Mochel).
Для больших систем, **sysfs** дорабатывалась *Манишем Сони* (Maneesh Sony).

В современных ОС Linux, *sysfs* важна также как и файловая система *proc*. Она монтируется в директорию */sys* буквально на первых этапах работы с временной файловой системой, во время загрузки ОС.

Всем зарегистрированным системой объектам инфраструктуры ядра ОС (упомянутым объектам ***kobjects***), в дирктории */sys* соответствуют свои каталоги.
Форма дерева каталогов, берущего свое начало в */sys*, максимально точно соответствует соотношению связей между объектами. Активно используемые символические ссылки делают это дерево каталогов достаточно разветвленным и отражающим отношения наследования, а также их связи между объектами.
Кроме того, проект *udev* имеет собственный язык описания устройств, который полностью заменяет язык HAL, использовавшийся ранее, и обеспечивает всему проекту большие перспективы развития, независимо от системы *systemd*.

В своей реализации, проект *udev* использует:

- *язык динамического управления* устройствами ЭВМ;
- *инфраструктуру файловой системы ОС*, где располагаются сценарии (правила) управления устройствами, а также другое ПО проекта.

Рассмотрим отдельно указанные компоненты проекта.

1.2.1 Язык динамического управления устройствами

Язык динамического управления устройствами не имеет официального названия и соответствующего сокращения.
Он представляет набор правил, обычно записываемых в текстовый файл, которые считывается демоном проекта *udev* и используются им при управлении устройствами, когда в системе возникают события, касающиеся этих устройств.
В целом, язык придерживается общих синтаксических норм, присущих языкам shell:

- символ «#» используется для коментариев, а также: *, ? и [] , как в bash;
- продолжение строки обозначается «\»;
- строковые значения заключаются в «**двойные кавычки**».

Каждое отдельное правило представляет собой отдельную строку, в которой, через запятую, перечисляются ключевые слова (ключи) двух видов:

- *ключи-условия* — это пары ***ключ/значение*** к которым применимы *операторы сравнения*;
- *ключи-присваивания* — это пары ***ключ/значение*** к которым применимы *операторы присваивания*;

Общий набор ключевых слов — ограничен. Рассмотрим их по отдельности вместе с используемыми операторами.

Ключи-условия описывают проверки, которые должны быть выполнены для применения к ним правил *udev*:

ACTION	- название события, связанного с устройством, например, добавление устройства ("add") или его удаление ("remove").
DEVPATH	- путь к каталогу (относительно файловой системы /sys), содержащему файлы event для данного устройства, например, для драйвера ipw3945 данный ключ примет следующее значение DEVPATH=/bus/pci/drivers/ipw3945.
KERNEL	- имя данное устройству ядром системы (внутреннее имя устройства).
SUBSYSTEM	- подсистема, в которой произошло возбуждение события, связанного с устройством, например, SUBSYSTEM="usb" для всех событий, связанных с USB-устройствами.
ATTR{filename}	- атрибуты устройства в файловой системе sysfs. Например, для нахождения строк, содержащих аргумент vendor (производитель устройства), можно использовать следующую строку ATTR{vendor}=="On[sS]tream".
KERNELS	- заставляет udev использовать DEVPATH для определения имени устройства.
SUBSYSTEMS	- заставляет udev использовать DEVPATH для определения подсистемы устройства.
DRIVERS	- заставляет udev использовать DEVPATH для определения драйвера устройства.
ATTRS{filename}	- заставляет udev использовать DEVPATH для устройства с указанным аргументом файловой системы sysfs.
ENV{key}	- ищет значение переменной окружения, например, ENV{ID_BUS}=="ieee1394" может использоваться для поиска всех событий, связанных с шиной FireWire.
PROGRAM	- заставляет udev выполнить внешнюю программу. В случае успешного завершения работы программа обязана возвращать код выхода равным нулю. Вывод программы доступен через ключ RESULT.
RESULT	- ищет соответствие выводу последнего вызова PROGRAM. Применяется либо в том же правиле, что и PROGRAM, либо в последующем.

Операторы ключей-условий:

«==»	- (два знака "равно" подряд без пробела) - проверка на равенство. Выражение справа знака равенства служит образцом для поиска.
«!=»	- (восклицательный знак и знак "равно" без пробела) - проверка на неравенство. Выражение справа знака равенства служит образцом для поиска.

Ключи-присваивания - присваивают значения, имена или какие-либо еще параметры файлам устройств, создаваемых *udev*:

NAME - имя создаваемого файла устройства.
Все последующие правила для данного устройства игнорируются.

SYMLINK - имя создаваемой символической ссылки на указанное устройство.
Для одного устройства может быть создано несколько символических ссылок одним (в этом случае список ссылок разделяется пробелами) или несколькими правилами udev.

OWNER, GROUP, MODE - задают соответствующие права доступа для создаваемого файла устройства.

ATTR{key} - задает параметр, который будет записан, как аргумент устройства в файловой системе sysfs.
Если используется оператор "==", то данный ключ также используется для поиска соответствия аргумента устройству.

ENV{key} - говорит udev экспортировать внешнюю переменную среды.
Если используется оператор "==" (см.выше), то данный ключ также используется для поиска соответствия указанной переменной среды.

RUN - просит udev добавить указанную программу в список программ, которые будут выполнены для данного устройства.
Для избежания блокирования дальнейших событий для этого устройства, необходимо чтобы данный список был как можно короче.

LABEL - метка для последующего перехода с помощью оператора GOTO.

GOTO - заставляет udev пропустить набор правил и продолжить со строки, указанной передаваемой GOTO меткой LABEL.

IMPORT{type} - загрузка переменных в окружение события, таких как, например, вывод внешней программы. udev может импортировать переменные нескольких различных типов.
Если тип не указан, то udev пытается самостоятельно его определить, ориентируясь на права доступа указанного файла:

- Если это исполняемый файл - udev выполняет внешнюю программу и импортирует ее вывод;
- Если это текстовый файл - udev импортирует его содержимое;
- Если это родительское устройство - udev импортирует его параметры.

WAIT_FOR_SYSFS - просит udev подождать создания указанного файла в файловой системе sysfs.
Например, WAIT_FOR_SYSFS = "ioerr_cnt" информирует udev подождать создание файла ioerr_cnt.

OPTIONS - данный ключ может принимать несколько возможных значений:

- last_rule заставляет udev игнорировать все последующие правила, касающиеся данного устройства;
- ignore_device заставляет udev игнорировать это событие полностью;
- ignore_remove заставляет udev игнорировать все последующие события удаления устройства;
- all_partitions заставляет udev создать файлы устройств для всех доступных разделов блочных устройств.

Операторы ключей-присваивания:

«=» - (знак "равно") - присваивание значения ключу.
Если данный ключ ранее содержал список значений, то этот список заменяется вновь присваиваемым значением.
Ключу можно присвоить только единственное значение.

«+=» - (знаки "плюс" и равно без пробела) - присвоить ключу, содержащему список записей еще одно значение.

«-=» - (знаки "минус" и равно без пробела) — удалить из ключа, содержащего список записей одно значение.

«:=» - (знаки "двоеточие" и равно без пробела) - присваивание ключу окончательного значения.
Запретить любые последующие изменения более поздними правилами.

Специальные обозначения в правилах *udev*, которые используются как и в обычных скриптах:

%r, $root - Директория для файлов устройств (по умолчанию /dev);

%p, $devpath - Значение DEVPATH;

%k, $kernel - параметр KERNEL или имя, присвоенное устройству ядром системы (другими словами, то имя под которым это устройство "знает" ядро;

%n, $number - номер устройства;

%N, $tempnode - временное имя файла устройства, которое создается для того, чтобы предоставить доступ к устройству до создания постоянного файла;

%M, $major - "старший" (major) номер устройства;

%m, $minor - "младший" (minor) номер устройства;

%s{attribute}, $attr{attribute} - значение атрибута файловой системы sysfs;

%E{variable}, $attr{variable} - значение переменной среды;

%c, $result - строка, возвращаемая внешней программой (PROGRAM). Также может быть выбрана часть строки (пробелы воспринимаются, как разделители строки), которая выделяется указанием номера части, как аргумента **%c{N}**. Если после числа стоит знак "+", то подставляется указанная часть строки плюс ее оставшаяся часть строки: **%c{N+}**;

%% - символ %;

$$ - символ $.

Далее, рассмотрим вопросы применения этого языка, привязав его к инфраструктуре проекта *udev*.

1.2.2 Инфраструктура проекта udev

Общая инфраструктура проекта *udev*, следующая:

/lib/systemd/systemd-udevd	- файл демона — менеджера устройств;
/bin/udevadm	- файл инструмента управления для *udev*;
/etc/udev/rules.d/	- директория с *наивысшим приоритетом*, для размещения файлов правил проекта *udev*; формируется администратором ОС;
/run/udev/rules.d/	- директория со *средним приоритетом*, для размещения файлов правил проекта *udev*; может формироваться динамически во время работы ОС;
/lib/udev/rules.d/	- директория с *низшим приоритетом*, для размещения файлов правил проекта *udev*; формируется при инсталляции пакета.

Имена файлов правил имеют общий формат:

```
    XX-<имя>.rules
```

где *XX* — две цифры, задающие упорядоченность файлов, используемое при просмотре правил;
<*имя*> - любая текстовая строка, формирующая имя файла.
Приоритет директорий используется при выборе файлов-правил с одинаковыми локальными именами.
Пример файла правил приведен на листинге 1.1.

Листинг 1.1 — Файл правил: /lib/udev/rules.d/70-mouse.rules

```
# do not edit this file, it will be overwritten on update

ACTION=="remove", GOTO="mouse_end"
KERNEL!="event*", GOTO="mouse_end"
ENV{ID_INPUT_MOUSE}=="", GOTO="mouse_end"

# mouse:<subsystem>:v<vid>p<pid>:name:<name>:*
KERNELS=="input*", ENV{ID_BUS}=="usb", \
        IMPORT{builtin}="hwdb 'mouse:\
$env{ID_BUS}:v$attr{id/vendor}p$attr{id/product}:name:$attr{name}:'", \
        GOTO="mouse_end"
KERNELS=="input*", ENV{ID_BUS}=="bluetooth", \
        IMPORT{builtin}="hwdb 'mouse:\
$env{ID_BUS}:v$attr{id/vendor}p$attr{id/product}:name:$attr{name}:'", \
        GOTO="mouse_end"
DRIVERS=="psmouse", SUBSYSTEMS=="serio", \
        IMPORT{builtin}="hwdb 'mouse:ps2::name:$attr{device/name}:'", \
        GOTO="mouse_end"

LABEL="mouse_end"
```

1.2.3 Утилита udevadm

Может показаться, что написание правил для *udev* является очень сложным или почти невозможным занятием. На самом деле – это не так. Необходимо лишь учитывать следующее:

- присутствует иерархия *KERNEL/SUBSYSTEM/DRIVER/ATTR* для четырех основных ключей;
- имеется утилита *udevadm*, входящая в инструментальный состав пакета *udev*;
- имеются достаточно подробные руководства ***man*** по *udev* и *udevadm*.

Для подтверждения сказанного, проведем беглый анализ устройства USB-мыши, которое наиболее легко может быть использовано в учебном процессе. Для этого, воспользуемся возможностями утилиты *udevadm*.

Чтобы получить ***подробную информацию*** об устройстве, следует воспользоваться командой:

```
udevadm info -a -n <Путь к файлу устройства>
```

Например, для устройства мыши, эта команда будет выглядеть так:

```
udevadm info -a -n /dev/input/mouse?
```

Начальная часть вывода этой команды показана на рисунке 2.1.

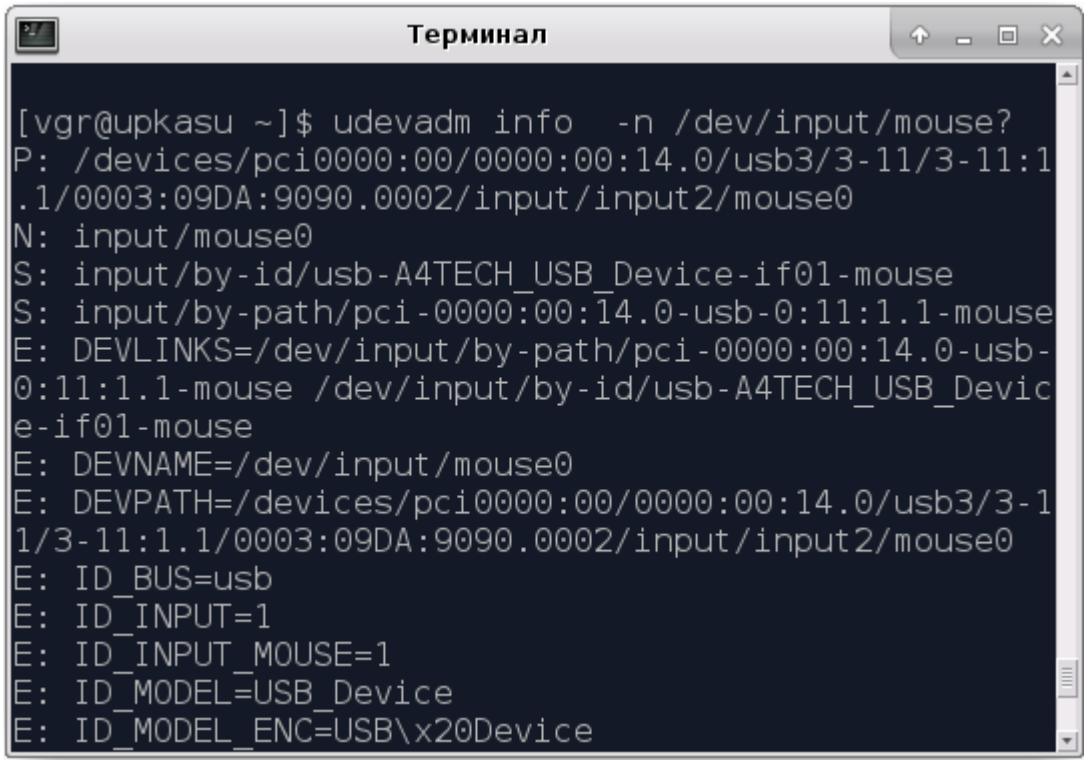

Рисунок 1.1 – Получение информации о характеристиках устройства мыши

Чтобы изучить, ***какие ключи используются*** при обработке событий устройства мыши, следует запустить *udevadm* в режиме монитора командой:

```
udevadm monitor --environment --udev
```

Монитор будет ждать событий, которые обрабатываются *systemd-udevd*.
Если ***подключить/удалить*** устройство мыши, то монитор покажет все обрабатываемые сообщения, как это отражено на рисунке 1.2.

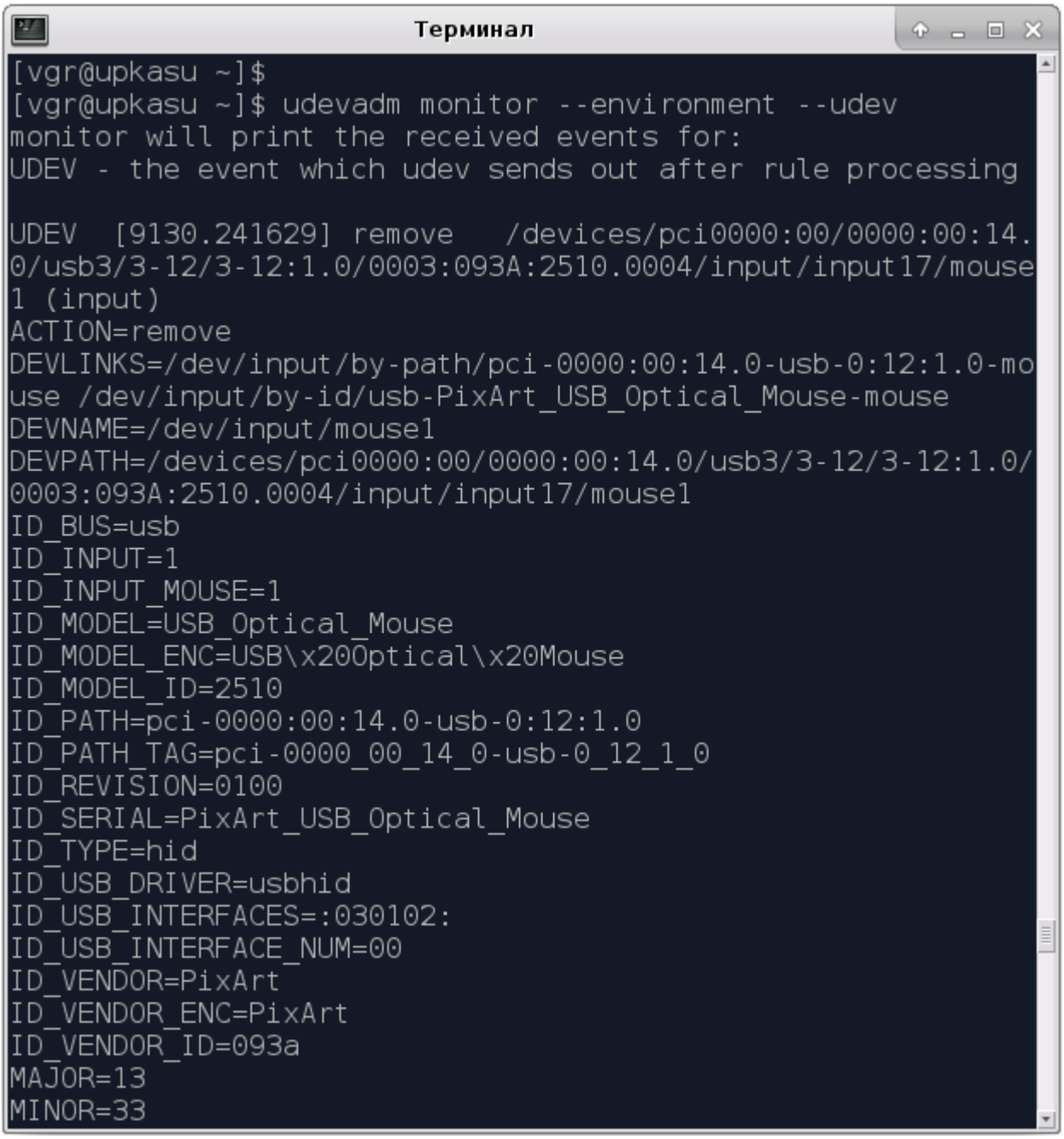

Рисунок 1.2 – Вывод монитора при удалении устройства мыши

Даже поверхностное исследование показывает, что:

- *ACTION=add*, когда мышь подключается к ЭВМ;
- *ACTION=remove*, когда мышь отключается от компьютера;

- *ID_INPUT_MOUSE=1* – идентифицирует устройство мыши;
- *DEVNAME=/dev/input/mouse?* - встречается только при одном событии, как при подключении мыши, так и при ее отключении.

На основании полученной информации, можно написать правила, которые будут запускать некоторую программу пользователя, когда мышь подключается или отключается.
Для примера, создадим файл правил *70-upk_mouse.rules*, текст которого приведен на рисунке 1.3.

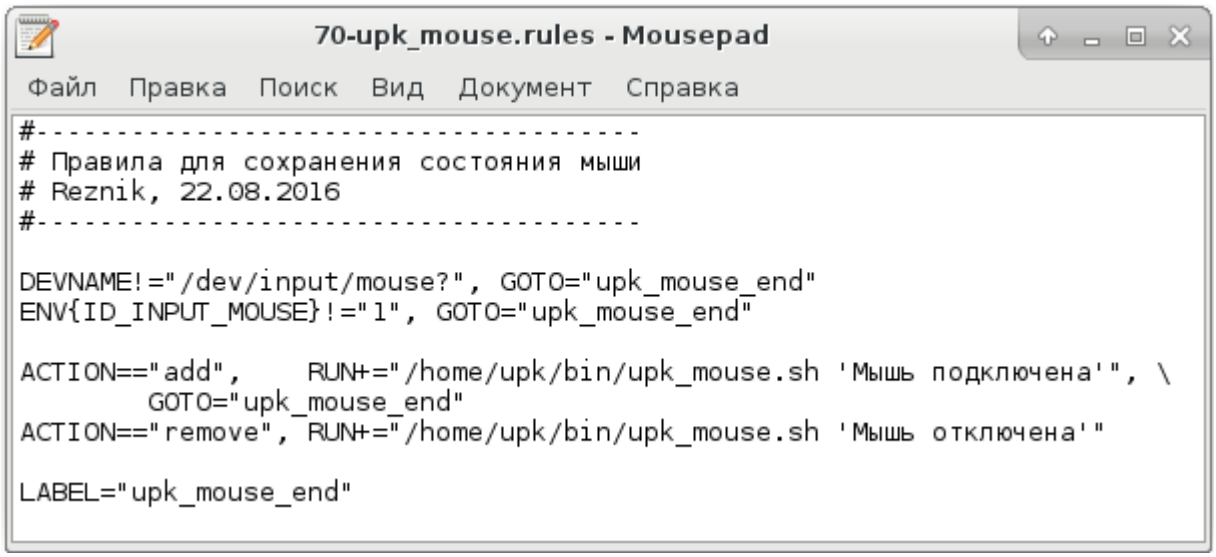

Рисунок 1.3 – Дополнительные правила устройства мыши

Если этот файл положить в директорию */etc/udev/rules.d*, то, при подключении или отключении мыши, он должен вызывать сценарий */home/upk/bin/upk_mouse.sh* с соответствующим параметром.
Если этот сценарий реализовать как показано на рисунке 1.4, то состояние мыши будет сохраняться в домашней директории пользователя *upk*, в файле *.upk_mouse*.

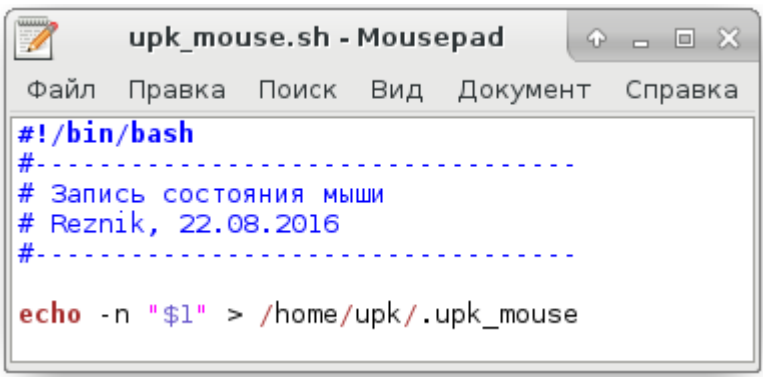

Рисунок 1.4 – Сценарий сохранения состояния устройства мыши

Теперь, если подключить или отсоединить устройство мыши, в домашней директории пользователя *upk*, файл *~/.upk_mouse* будет содержать запись о ее состоянии. Студент может убедиться в этом, при выполнении лабораторных работ.

Утилита *udevadm* имеет и другие полезные возможности, например, такую как отладка сценария правил.
Применительно к нашему примеру, запустим ее в варианте тестирования:

```
udevadm test –a add /sys/class/input/mouse?
```

В таком варианте, она покажет имитацию последовательности просмотра всех файлов правил, как если бы реально поступил сигнал ACTION=="add".
В частности, если правило обращается к внешней функции, как в нашем случае, то будет указан и сам вызов, заданный ключевым словом RUN.
С другой стороны, утилита не будет выполнять реальные действия, а только эмитировать их, что позволит, например, убедиться *в наличии или отсутствии* вызова внешеней функции интересующим нас правилом.

Замечание

Обычно, ***systemd-udevd*** отслеживает наличие и изменение файлов правил.
Чтобы принудительно заставить систему перезагрузить все правила, следует воспользоваться одной из команд:

```
udevadm control --reload-rules
udevadm trigger
```

Как было отмечено ранее, проект *udev* интенсивно использует псевдофайловые системы *proc* и *sysfs*, к изучению которых мы и переходим в следующих подразделах данной темы.

1.3 Псевдофайловая система proc

Современные операционные системы, на примере менеджера устройств *udev*, достаточно интенсивно используют псевдофайловые системы ***procfs*** и ***sysfs***, которые монтируются в директории */proc* и */sys*, соответственно.
В данном подразделе будет рассмотрена только файловая система ***procfs***, которая в ОС UNIX появилась еще *в середине 80-х годов* прошлого столетия, но использовалась только на достаточно мощных компьютерах серверного исполнения, поскольку создается в оперативной памити ЭВМ (ОЗУ).

Замечание

Файловая система ***procfs*** является стандартной принадлежностью всех UNIX систем, таких как Free/Open/Net BSD, Solaris, и других, а ее наличие и принципы использования оговариваются стандартом POSIX 2.
Файловая система ***sysfs*** является «порождением» ОС Linux и используется только в этой системе.

Файловая система *procfs* появилась как следствие реализации основной парадигмы ОС UNIX - «*Все есть файл!*», что в данном контексте означает:

- *наличие диагностического канала* передачи информации из ядра ОС в пространство пользователя;
- *наличие управляющего канала* передачи информации из пространства пользователя в ядро ОС.

Фактически, указанные каналы призваны полностью заменить потребность в системных вызовах *ioctl()*, которые уже давно считаются *устаревшими и потенциально опасными*, по причине отсутствия контроля типизации передаваемых данных.
Сама файловая система монтируется в директорию */proc*, буквально в первых командах работы с временной файловой системой, а затем обязательно переносится в корневую файловую систему.

Формирование структуры имен, в ФС *procfs*, обеспечивается модулями ядра ОС. Основную работу, по созданию и уничтожению имен в */proc*, выполняют четыре вызова, определенные в файле **<*linux/proc_fs.h*>** (см. листинг 1.2):

Листинг 1.2 — Системные вызовы создания и удаления имен в /proc

```
struct proc_dir_entry *proc_mkdir_mode( const char *name, umode_t mode,
                                        struct proc_dir_entry *parent );
int *remove_proc_subtree( const char *name, struct proc_dir_entry *parent );

struct proc_dir_entry *proc_create( const char *name, umode_t mode,
   struct proc_dir_entry *parent, const struct file_operations *proc_fops );
void remove_proc_entry( const char *name, struct proc_dir_entry *parent );
```

где *name* — указатель на строку, содержащую *имя файла*;
mode — параметры доступа к *имени файла*;

parent — указатель на достаточно большую структуру типа *proc_dir_entry*, которую в современных ОС найти достаточно трудно; она соответствует *родительской директории* в */proc* для создаваемого файла; если *parent=NULL*, то имя будет создаваться в самой директории */proc*;
proc_fops — указатель на структуру типа *file_operations*, которую мы уже рассматривали в предыдущей теме и ссылка на которую содержится в *parent*.

Возможности указанных функций ядра следующие:

- *proc_mkdir_mode()* - создает директории в */proc*, с заданными правами доступа;
- *remove_proc_subtree()* - удаляет директории из */proc*;
- *proc_create()* - создает файлы в */proc*, с заданными правами доступа;
- *remove_proc_entry()* - обеспечивает удаление файлов и директорий из */proc*.

Таким образом, использование этих функций ядра позволяет любому модулю создавать в */proc* произвольную иерархию файлов, доступных на чтение и запись с заданными правами.

Чтобы продемострировать указанные свойства файловой системы *procfs*, а затем закрепить знания о них на практике, рассмотрим конкретный пример, исходный код которого представлен на листинге 1.3, обеспечивающий:

- создание в директории */proc* учебной каталога *upk_directory*;
- создание в директории */proc/upk_directory* двух файлов с именами *file1* и *file2*, доступных как на чтение, так и на запись.

Листинг 1.3 — Пример работы с файлами директории /proc

```
/*
 *  upk-proc.c: Модуль работы с файловой системой procfs
 *  Модифицировано: Reznik, 24.08.2016
 */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>    /* для put_user */
#include <linux/proc_fs.h>  /* API для работы с procfs */
/*
 *  Прототипы
 */
int init_module(void);
void cleanup_module(void);
static int proc_open1(struct inode     *, struct file *);
static int proc_release1(struct inode  *, struct file *);
static ssize_t proc_read1(struct file  *, char *, size_t, loff_t *);
static ssize_t proc_write1(struct file *, const char *, size_t, loff_t *);
static int proc_open2(struct inode     *, struct file *);
static int proc_release2(struct inode  *, struct file *);
static ssize_t proc_read2(struct file  *, char *, size_t, loff_t *);
static ssize_t proc_write2(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define NAME_DIR "upk_directory" /* Имя директории, добаляемое в /proc  */
#define FNAME1 "file1" /* Имя файла, добаляемое в /proc/upk_directory   */
```

```
#define FNAME2 "file2" /* Имя файла, добаляемое в /proc/upk_directory   */
#define BUF_LEN 1024   /* Максимальная длина сообщения в файле */
#define MSG_ERR "=== Не могу зарегистрировать модуль upk-proc.ko\n"
/*
 * Глобальные переменные, декларированные как static, глобальные внутри модуля.
 */
static int f_open1 = 0; /* Используется, чтобы ограничить
                         * множественный доступ к файлу file1*/
static int f_open2 = 0;

static char buff1[BUF_LEN + 1];   /* Буфер для первого файла */
static char buff2[BUF_LEN + 1];   /* Буфер для первого файла */

/*
 * Переопределение функций: read(), ..., release()
 */
static struct file_operations fops1 = {
      .owner   = THIS_MODULE,
      .read    = proc_read1,
      .write   = proc_write1,
      .open    = proc_open1,
      .release = proc_release1
};
static struct file_operations fops2 = {
        .owner   = THIS_MODULE,
        .read    = proc_read2,
        .write   = proc_write2,
        .open    = proc_open2,
        .release = proc_release2
};
/*
 * Структуры для файлов procfs.
 */
static struct proc_dir_entry * pdir; /* Указатель для директории*/
static struct proc_dir_entry * pf1;  /* Указатель для file1*/
static struct proc_dir_entry * pf2;  /* Указатель для file2*/

/*
 * Эта функция вызывается, когда модуль загружается.
 */
int init_module(void)
{
   pdir = proc_mkdir_mode(NAME_DIR, 0555, NULL);
   if(pdir == NULL) {
     printk( KERN_ERR MSG_ERR );
     return -ENOMEM;
   }

   pf1 = proc_create(FNAME1, 0666, pdir, &fops1);
   if(pf1 == NULL) {
     remove_proc_entry(NAME_DIR, NULL);
     printk( KERN_ERR MSG_ERR );
     return -ENOMEM;
   }

   pf2 = proc_create(FNAME2, 0666, pdir, &fops2);
   if(pf1 == NULL) {
     remove_proc_entry(FNAME1,   pdir);
     remove_proc_entry(NAME_DIR, NULL);
     printk( KERN_ERR MSG_ERR );
```

```
      return -ENOMEM;
   }

   printk(KERN_INFO "Модуль upk-proc.ko - стартовал!\n");
   buff1[0] = '\0';
   buff2[0] = '\0';
   return 0;
}

/*
 * Эта функция вызывается, когда драйвер выгружается.
 */
void cleanup_module(void)
{
        /*
         * Unregister the device
         */
      printk(KERN_INFO "Модуль upk-proc.ko - завершил работу!\n");
      remove_proc_entry(FNAME1,   pdir);
      remove_proc_entry(FNAME2,   pdir);
      remove_proc_subtree(NAME_DIR, NULL);
}

/*
 * Методы:
 *
 * Вызывается, когда процесс открывает файл, подобно:
 * "cat /proc/upk_proc/file1"
 */
static int proc_open1(struct inode *inode, struct file *file)
{
        static int count1 = 0;

        if (f_open1)
                return -EBUSY;

        f_open1++;
        printk("Устройство %s открывается %d-й раз!\n",
           FNAME1, ++count1);
        try_module_get(THIS_MODULE);

        return SUCCESS;
}

static int proc_open2(struct inode *inode, struct file *file)
{
        static int count2 = 0;

        if (f_open2)
                return -EBUSY;

        f_open2++;
        printk("Устройство %s открывается %d-й раз!\n",
           FNAME2, ++count2);
        try_module_get(THIS_MODULE);

        return SUCCESS;
}

/*
```

```
 * Вызывается, когда процесс закрывает файл устройства.
 */
static int proc_release1(struct inode *inode, struct file *file)
{
        f_open1--;          /* Мы теперь готовы для следующего вызова */

        module_put(THIS_MODULE);

        return 0;
}

static int proc_release2(struct inode *inode, struct file *file)
{
        f_open2--;          /* Мы теперь готовы для следующего вызова */

        module_put(THIS_MODULE);

        return 0;
}

/*
 * Вызывается, когда процесс, который уже открыл файл устройства,
 * пытается читать из него.
 */
static ssize_t proc_read1(struct file *filp, /* смотри include/linux/fs.h */
                          char *buffer,     /* буфер для данных */
                          size_t length,    /* длина буфера     */
                          loff_t * offset)  /* Смещение от начала файла */
{
  static int finish1 = 0;
  int i;

  /*
   * Для индикации признака конца файла возвращается 0.
   * Если этого не сделать, процесс будет продолжать пытаться читать из
   * файла, угодив в бесконечный цикл.
   */
  if (finish1) {
     finish1 = 0;
     return 0;
  }

  /*
   * Для передачи данных из пространства ядра в пространство пользователя
   * следует использовать put_user. В обратном направлении - get_user.
   */
  for (i = 0; i < length && buff1[i]; i++)
     put_user(buff1[i], buffer + i);

  /*
   * Обратите внимание: в данной ситуации мы исходим из предположения,
   * что размер сообщения меньше, чем lenght, в противном случае
   * - сообщение будет обрезано!
   * В реальной ситуации, если длина сообщения больше чем lenght,
   * то остаток сообщения возвращается на последующих вызовах.
   */
  finish1 = 1;

  return i;             /* Вернуть количество "прочитанных" байт */
}
```

```
static ssize_t proc_read2(struct file *filp, /* смотри include/linux/fs.h */
                          char *buffer,     /* буфер для данных */
                          size_t length,    /* длина буфера     */
                          loff_t * offset)
{
  static int finish2 = 0;
  int i;

  if (finish2) {
     finish2 = 0;
     return 0;
  }

  for (i = 0; i < length && buff2[i]; i++)
     put_user(buff2[i], buffer + i);

  finish2 = 1;

  return i;             /* Вернуть количество "прочитанных" байт */
}

/*
 * Вызывается, когда процесс записывает в файл устройства, например:
 *  echo "hi" > /proc/upk_directory/file1
 */
static ssize_t
proc_write1(struct file *filp, const char *buff, size_t len, loff_t * off)
{
        int i;

     /*
      * Переместить данные, полученные от пользователя в буфер,
      * который позднее будет выведен функцией proc_read1().
      */
     for (i = 0; i < BUF_LEN && i < len; i++)
          get_user(buff1[i], buff + i);

     buff1[i] = '\0';    /* Обычная строка, завершающаяся символом \0 */
     return i;
}

static ssize_t
proc_write2(struct file *filp, const char *buff, size_t len, loff_t * off)
{
        int i;

     /*
      * Переместить данные, полученные от пользователя в буфер,
      * который позднее будет выведен функцией proc_read2().
      */
     for (i = 0; i < BUF_LEN && i < len; i++)
          get_user(buff2[i], buff + i);

     buff2[i] = '\0';    /* Обычная строка, завершающаяся символом \0 */
     return i;
}
```

1.4 Псевдофайловая система sysfs

Одно из современных достижений, — это появление *единой унифицированной модели представления устройств* в Linux. Оно появилось в ядре ОС, начиная с версии 2.6.
Основные составляющие данной модели - это файловая система *sysfs*, которая монтируется в директорию */sys*, и поддерживаемый ею пакет пользовательского пространства *udev*.
Модель устройств — это единый механизм для представления устройств и описания их топологии в системе.

Создание единого представления устройств рассчитано на получение следующих преимуществ:

- *уменьшение* дублирования кода;
- *использование* стандартного механизма для выполнения общих, часто встречающихся функций, таких как счетчики использования;
- *возможность систематизации* всех устройств в системе;
- *возможность просмотра* состояний устройств и определения, к какой шине то или другое устройство подключено;
- *обеспечение возможности* связывания устройств с их драйверами и наоборот;
- *возможность разделения* устройств на категории, в соответствии с различными классификациями, без знания физической топологии устройств;
- *обеспечение возможности* просмотра иерархии устройств от листьев к корню и выключение питания устройств в правильном порядке.

Методы работы с файловой системой - достаточно разнообразны.
Мы ограничимся изучением только одного из них, который базируется на понятии модели представления объектов устройств, являющимися объектами *struct kobject*, определенными в файле <***linux/kobject.h***>.
Обычно, сами *kobject* не используются напрямую, а входят в объекты ядра более высокого уровня.
Мы изучим работу только с объектами типа *class*, которые обеспечивают наиболее простой способ взаимодействия с модулями и драйверами ядра ОС.

Основная идея использования технологии объектов *class* заключается в следующем:

- *создается объект* типа *class*, имя которого отображается ***как каталог*** в директории */sys/class*;
- *для объекта создаются атрибуты*, необходимое количество и состав которых определяется замыслом разработчика, имена которых становятся именами файлов в директории класса, а их значения — содержимым этих файлов;
- *каждый атрибут имеет две свои функции*: *show()* - для чтения атрибута и *store()* - для установки значения (естественно, с учетом прав доступа).

Инструментальные возможности этой идеи, в основном, отражены в заголовочном файле <***linux/device.h***>.

Класс создается макросом и удаляется функцией:

```
    struct class *cls = create_class(THIS_MODULE, const char *name);
    void class_destroy(struct class *cls);
```

где *cls* — указатель на структуру типа *class* (см. <***linux/device.h***>);
name — имя создаваемого класса.

Атрибуты создаются и удаляются функциями:

```
int  class_create_file(struct class *cls, const struct class_attribute *attr);
void class_remove_file(struct class *cls, const struct class_attribute *attr);
```

где *cls* — структура, соответствующего класса;
attr — указатель на структуру типа *class_attribute*, содержание которой показано на листинге 1.4.

Листинг 1.4 — Структура class_attribute

```
struct class_attribute {
   struct attribute attr;
   ssize_t (*show)(struct class *class, struct class_attribute *attr, char *buf);
   ssize_t (*store)(struct class *class, struct class_attribute *attr,
                    const char *buf, size_t count);
};
```

Обычно, структура *class_attribute* создается макросом:

```
#define CLASS_ATTR( _name, _mode, _show, _store) \
     struct class_attribute class_attr_##_name = \
     __ATTR( _name, _mode, _show, _store)
```

где *_name* — имя атрибута;
_mode — права доступа к атрибуту;
_show — имя функции, читающей значение атрибута;
_store — имя функции, записывающей значение атрибута.

Чтобы более наглядно продемонстрировать возможности этой технологии, рассмотрим пример модуля, приведенного на листинге 1.5, который:

- создает класс с именем *upk-class*, отображаемый в директории */sys/class*;
- для этого класса, создаются три атрибута с именами: *data1*, *data2* и *data3*.

Замечание

Права доступа *_mode* следует выставить в одно из значений ***0600***, ***0660*** или ***0664***, иначе модуль не сможет откомпилироваться.

Листинг 1.5 — Исходный текст модуля upk-class.c

```
/*
 *  upk-class.c: Модуль для работы с файловой системой sysfs
 *  Модифицировано: Reznik, 24.08.2016
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/device.h>
#include <asm/uaccess.h>

#define LEN_MSG 180
static struct class *upk_class;

// Макрос для функций-обработчиков
// Обязательно: mode=0660 или 0664, иначе не откомпилируется!!!
#define IOFUNCS( name )                                              \
static char buf_##name[ LEN_MSG + 1 ] =                              \
               "Начальное значение "#name"\n";                       \
static ssize_t show_##name( struct class *class,                     \
                   struct class_attribute *attr,                     \
                            char *buf ) {                            \
   strcpy( buf, buf_##name );                                        \
   return strlen( buf );                                             \
}                                                                    \
static ssize_t store_##name( struct class *class,                    \
                    struct class_attribute *attr,                    \
                             const char *buf, size_t count ) {       \
   if (count > LEN_MSG) count = LEN_MSG;                             \
   printk( "Записано %d байта в "#name"\n", (int)count );            \
   strncpy( buf_##name, buf, count );                                \
   buf_##name[ count ] = 0;                                          \
   return count;                                                     \
}

IOFUNCS( data1 );
IOFUNCS( data2 );
IOFUNCS( data3 );

// Определение атрибутных файлов
#define UPK_CLASS_ATTR( name )              \
struct class_attribute class_attr_##name = \
     __ATTR( name, 0664, &show_##name, &store_##name )

static UPK_CLASS_ATTR( data1 ); // Для создания class_attr_data1
static UPK_CLASS_ATTR( data2 ); // Для создания class_attr_data2
static UPK_CLASS_ATTR( data3 ); // Для создания class_attr_data3

int __init upk_init(void) {
   int res;
   upk_class = class_create( THIS_MODULE, "upk-class" );
   if( IS_ERR( upk_class ) )
```

```
     printk( "Ошибка открытия класса upk-class\n" );

   res = class_create_file( upk_class, &class_attr_data1 );
   if( res )
     printk( "Ошибка открытия класса data1\n" );

   res = class_create_file( upk_class, &class_attr_data2 );
   if( res )
     printk( "Ошибка открытия класса data2\n" );

   res = class_create_file( upk_class, &class_attr_data3 );
   if( res )
     printk( "Ошибка открытия класса data3\n" );

   printk( "Модуль upk-class.ko - инициализирован...\n" );
   return 0;
}

void upk_cleanup(void) {
   class_remove_file( upk_class, &class_attr_data1 );
   class_remove_file( upk_class, &class_attr_data2 );
   class_remove_file( upk_class, &class_attr_data3 );
   class_destroy( upk_class );
   printk( "Модуль upk-class.ko - закончил работу...\n");
   return;
}

module_init( upk_init );
module_exit( upk_cleanup );
MODULE_LICENSE( "GPL" );
```

Созданный на основе этого листинга модуль позволяет:

- загружаться и выгружаться от имени пользователя *root*;
- обычный пользователь, например *upk*, может только читать из файлов *data1*, *data2* и *data3*, располагающихся в директории */sys/class/upk-class*, после загрузки модуля;
- пользователь *root* может как читать из этих файлов, так и писать в них;
- после выгрузки модуля, указанные директория и файлы, - удаляются.

Например, чтение из файла *data2* можно осуществить командой:

```
    cat /sys/class/upk-class/data2
```

Запись в этот файл можно осуществить командой:

```
    echo "Всем привет\!\!\!" > /sys/class/upk-class/data2
```

2 Лабораторная работа №3

Тема: Управление устройствами и псевдофайловые системы

Выполнение данной лабораторной работы предполагает, что студент изучил теоретический материал, изложенный в разделе 1 данного учебного пособия.

Общая цель лабораторной работы — на примере ОС Linux получить начальные практические навыки взаимодействия с устройствами ЭВМ, отображаемых в специальных файловых системах типа *devtmpfs*, *procfs* и *sysfs*.

Общие организационные вопросы, связанные с выполнением лабораторной работы, предполагают:

- успешную загрузку студентом ОС УПК АСУ, подключение личного архива на flshUSB по теме *sos*, а также переход в среду пользователя *upk*;
- создание в домашней директории каталога *~/lab3_sos*, необходимого для локализации результатов выполненых работ;
- следование заданиям, изложенным в подразделах 2.1, 2.2 и 2.3;
- проведение учебных работ, с обязательным отражением их результатов в индивидуальных отчетах.

Если перейти в директорию */dev/disk*, то мы увидим набор каталогов, как показано на рисунке 2.1.

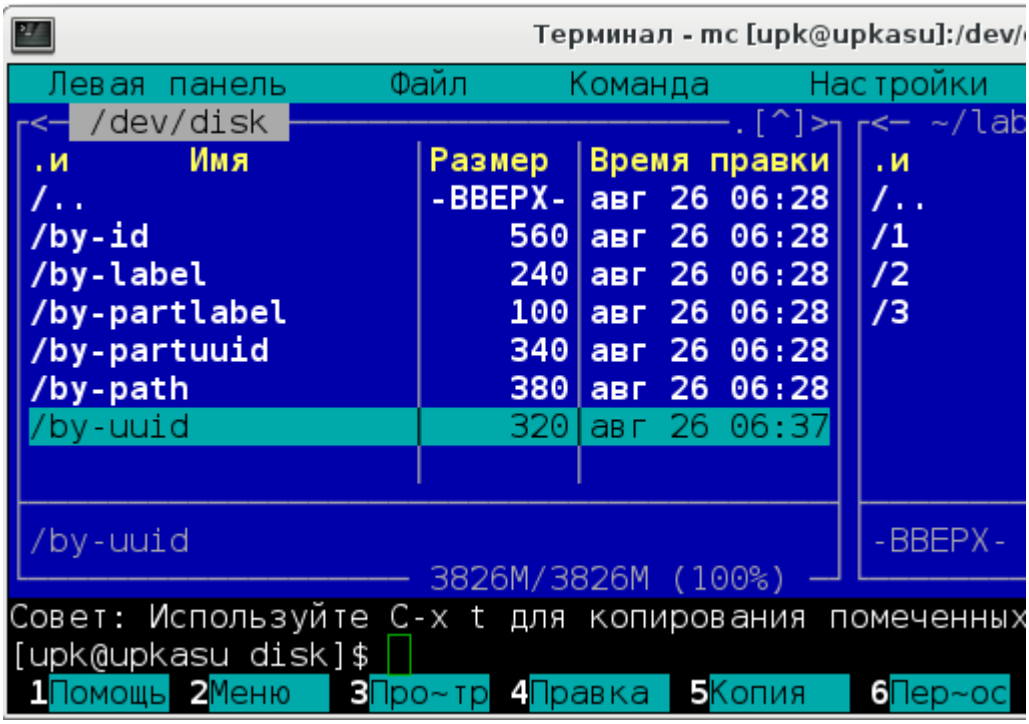

Рисунок 2.1 — Каталоги директории /dev/disk

Если воспользоваться утилитой ***tree*** и выполнить команду:

```
    tree /dev/disk
```

то мы увидим содержимое этих каталогов, которое, как показано на рисунке 2.2, представляет собой набор идентификаторов, являющихся ссылками на файлы устройств, расположенных в директории */dev*.

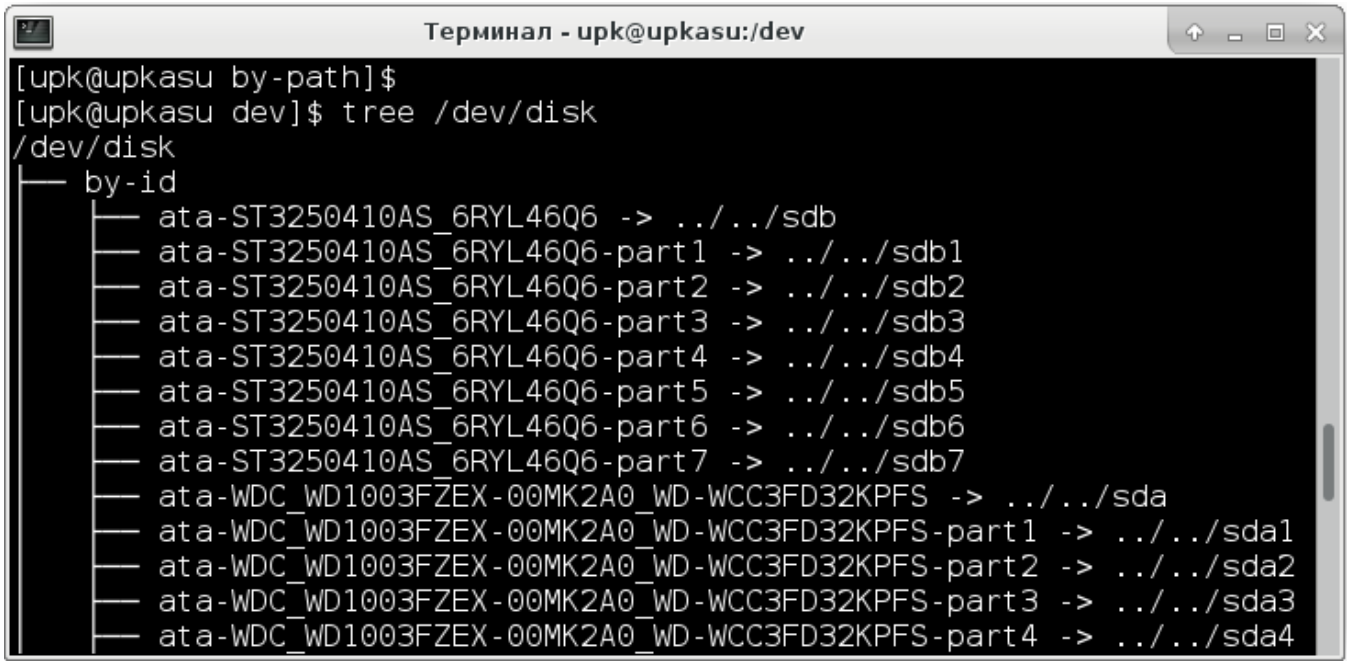

Рисунок 2.2 — Использование утилиты tree

Естественно, возникают вопросы: *кто, как и зачем* создал такое дерево файлов?
Ответ — однозначный: программное обеспечение проекта ***udev***, опирающееся на возможности современных псевдофайловых систем: *devtmpfs*, *procfs* и *sysfs*.

Практическая часть выполняемой лабораторной работы основана на закреплении теоретических знаний, полученных при изучении теоретической части данной темы, посредством реализации трех проектов, задания для которых представлены в подразделах 2.1 — 2.3.

Предполагается, что при выполнении конкретных заданий, требующих создание модулей ядра ОС, студент будет самостоятельно использовать навыки, полученные при выполнении лабораторной работы №2 [5].

2.1 Использование языка динамического управления устройствами

Основную работу по управлению файлами устройств выполняет ПО проекта *udev*, которое ***с 2012 года*** входит в состав проекта *systemd*, о чем мы будем говорить в следующей теме.
Главное, серверный сервисный процесс *systemd-udevd* отслеживает все события, связанные с изменением состояния устройств, что отражается появлением файлов и ссылок в псевдофайловых системах, монтированных к директориям: */dev*, */proc* и */sys*.

Для практического освоения технологии управления устройствами, проведем учебное исследование - на примере устройства USB-мыши, которое легко осуществить в учебных целях, без опасности нарушения работы ОС, в целом.
Методическое обеспечение такого исследования предполагает, что студент:

- изучил теоретическую часть методического пособия, изложенную в подразделе 1.2, а также имеет на компьютере устройство мыши (любого типа);
- запустил ОС УПК АСУ и успешно подключил тему обучения;
- находится в системе, зарегистрировавшись пользователем *upk*;
- создал директорию *~/lab3_sos/1*, для размешения и сохранения результатов исследования.

Основная идея исследования состоит в том, что, когда к системе (компьютеру) подключается или отключается устройство мыши, этот факт должен быть отражен в виде соответствующей записи в файле: *~/.upk_mouse*.

Реализацию такого исследования проведем в три этапа.
Этап 1 включает:

- *создание* в директории *~/lab3_sos/1* - файла *70-upk_mouse.rules*, например так, как показано на рисунке 1.3 (стр. 16), а затем — файла *upk_mouse.sh*, как показано на рисунке 1.4 (стр. 16);
- *файл 70-upk_mouse.rules* следует скопировать пользователем *root* - в директорию */etc/udev/rules.d*, а файл *upk_mouse.sh* — пользователем *upk*, - в директорию *~/bin*;
- *сделать* файл *upk_mouse.sh* — исполняемым.

Этап 2 состоит в запуске одной из команды:

```
udevadm control --reload-rules
udevadm trigger
```

обеспечивающей перезагрузку демоном *systemd-udevd* правил системы.

Этап 3 состоит из двух действий:

- *подключению* к компьютеру и *отключению* от него устройства мыши;

- *отслеживание* содержимого файла *~/.upk_mouse*, в процессах выполнения первого действия.

Проверить состояние мыши можно непосредственным просмотром содержимого файла *~/.upk_mouse* или командой:

```
cat ~/.upk_mouse
```

Замечание

Содержимое файла *~/.upk_mouse* отражает только факты изменений состояния некоторого устройства мыши, поэтому:

- если в момент включения компьютера мышь уже была подключена к компьютеру, то это не отразится на содержании указанного файла;
- если перед отключением темы пользователя *upk* в этот файл записать неправильное значение, то оно останется и после последующего запуска компьютера.

Данная ситуация связана с тем, что:

- сами правила написаны только на реакцию изменения состояния мыши;
- в момент включения компьютера, демон *systemd-udevd* запускается дважды: после загрузки ядра, когда он работает с временной файловой системой и создает в директории */dev* файлы устройств, и после монтирования корневой файловой системы, когда запускается основной управляющий процесс *systemd*.

Решение проблемы актуального отображения состояния устройства мыши, обозначенной в предыдущем замечании, можно решить например так:

- *учесть*, что подключенные к системе устройства мыши отображаются в директории */dev/input* как файлы *mouse0*, *mouse1* и так далее;
- *написать сценарий* языка shell, например — *test_mouse.sh*, как показано на рисунке 2.3, который следует поместить в директорию *~/bin*, сделать его исполняемым и протестировать посредством запуска сценария в терминале.

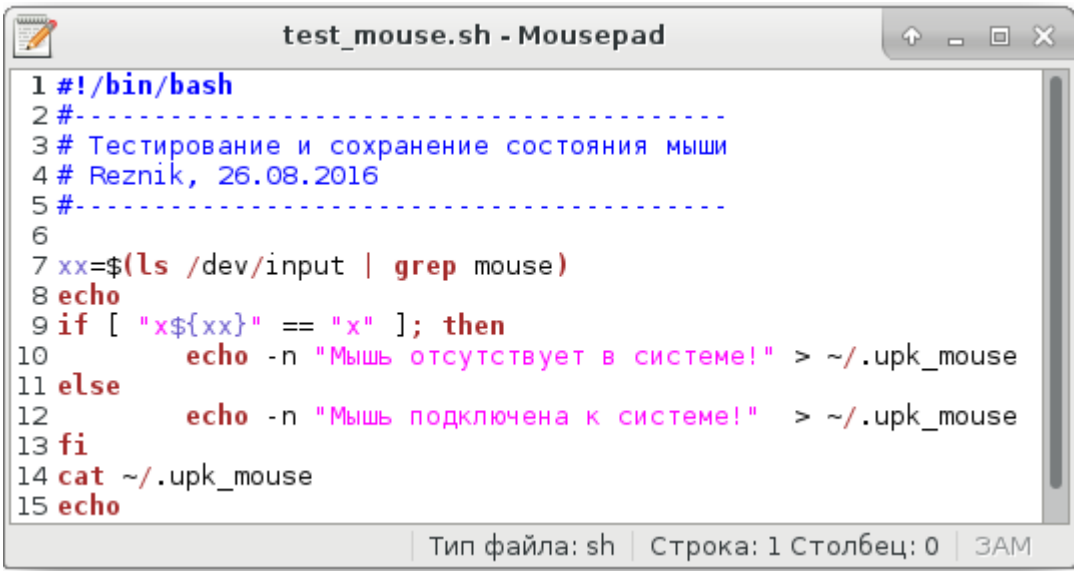

Рисунок 2.3 — Текст сценария для тестирования подключения мыши

Если теперь вызов сценария *test_mouse.sh* поместить в сценарий *~/bin/Attantion*, как показано на рисунке 2.4, то тестирование состояния устройства мыши будет происходить на этапе входа в сессию пользователя и отбражаться, как показано на рисунке 2.5.

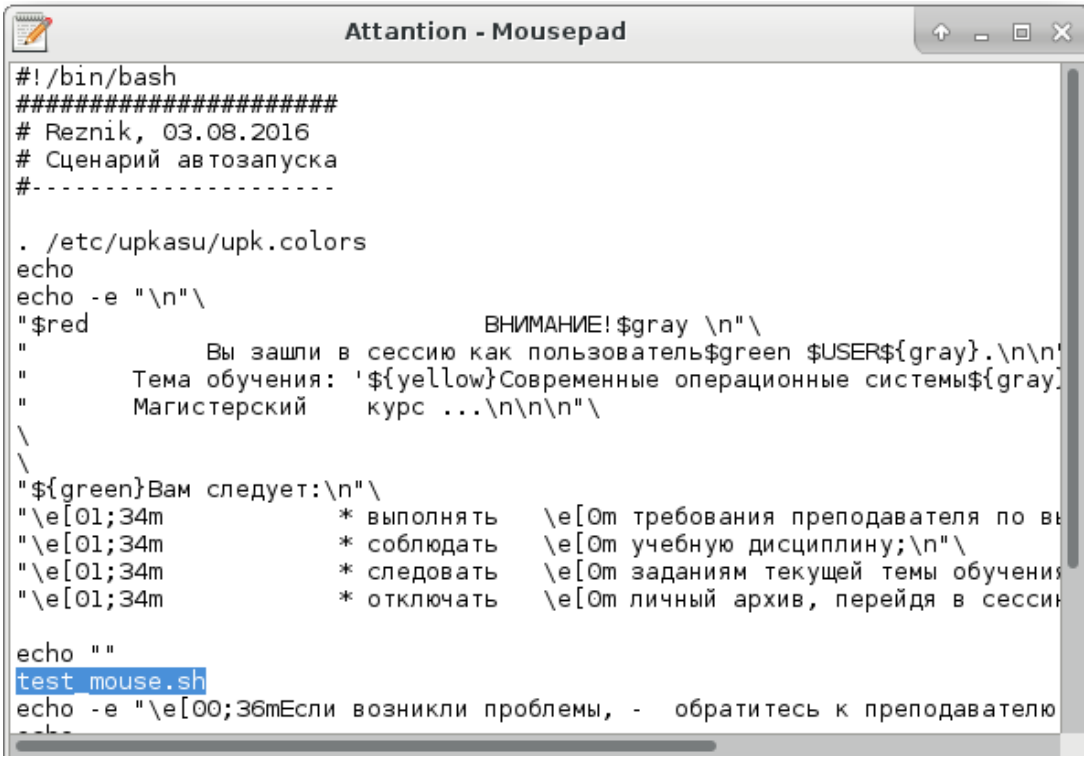

Рисунок 2.4 — Вставка вызова сценария в файл ~/bin/Attantion

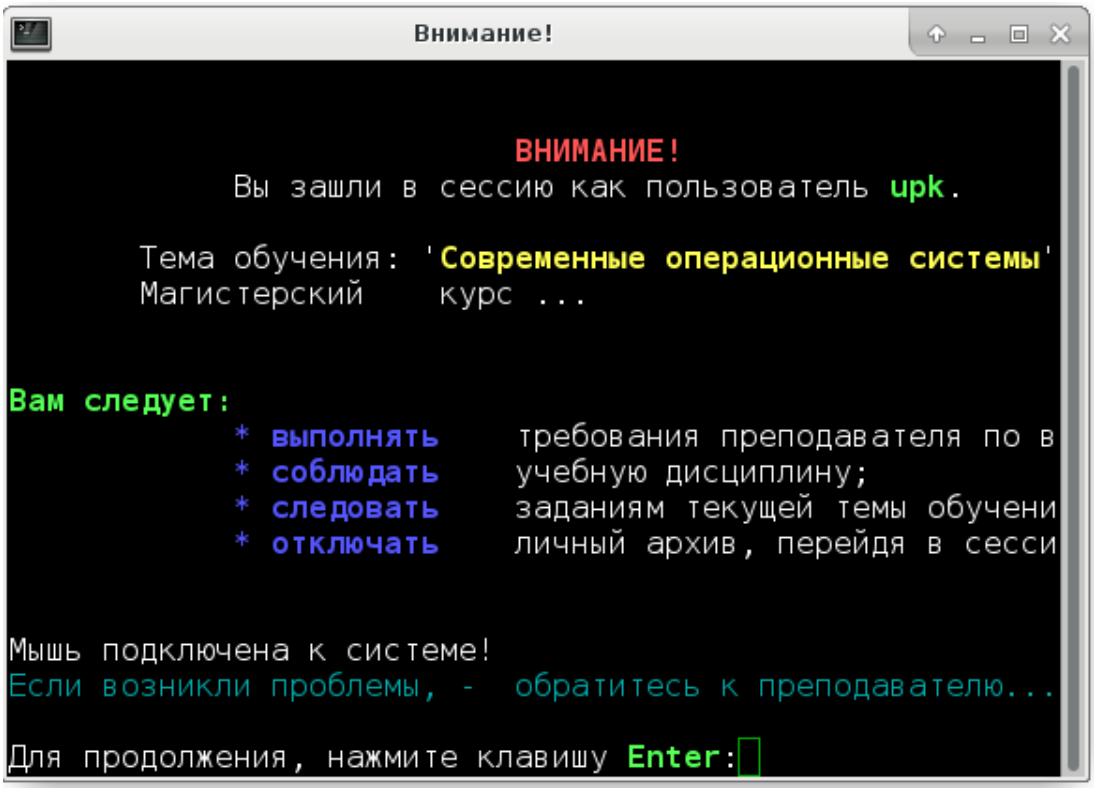

Рисунок 2.5 — Тестирование состояния мыши после входа в систему

2.2 Модуль для работы с псевдофайловой системой procfs

Исследования и разработки псевдофайловой системы *procfs* стали вестись со второй половины 80-х годов.
Цель таких работ — обеспечение доступа к информации ядра ОС, минуя системные вызовы *ioctl()*, которые:

- *не имеют стандарта* на состав и количество необходимых вызовов;
- *требуют создания отдельного приложения* (утилиты), которая обслуживала бы эти вызовы;
- *требуют создания сопроводительной документации*, поясняющей правила запуска этой утилиты, а также интерпретирующей ее вывод.

Для практического освоения технологии получения информации из ядра ОС, проведем учебное исследование — на примере реализации модуля, который создает в файловой системе *procfs* *одну директорию* и *два файла*, доступных как для чтения, так и для записи текстовой информации.

Методическое обеспечение такого исследования предполагает, что студент:

- изучил теоретическую часть методического пособия, изложенную в подразделе 1.3;
- запустил ОС УПК АСУ и успешно подключил тему обучения;
- находится в системе, зарегистрировавшись пользователем *upk*;
- создал директорию *~/lab3_sos/2*, для размещения и сохранения результатов исследования.

Основная цель данного учебного исследования — закрепление знаний по применению специального системного ПО ОС, необходимого для работы с файловой системой *procfs*.

Реализация проекта по созданию модуля состоит в следующем:

- в директории *~/lab3_sos/2*, создается текстовый файл *upk-proc.c*, в который переносится содержимое листинга 1.3 (стр. 19);
- в директории *~/lab3_sos/2*, создается текстовый файл *Makefile*, в котором содержатся операторы, необходимые для создания модуля;
- с помощью утилиты *make*, от имени пользователя *root*, проводится создание модуля *upk-proc.ko*.

Замечание

Технология создания модуля *upk-proc.ko* является такой же как и при выполнении лабораторной работы №2.
Чтобы создать файл модуля, следует:

- открыть терминал и запустить файловый менеджер командой: *sudo mc* ;
- перейти в директорию *~/lab3_sos/2* и выполнить команду: *make* .

Загрузив модуль командой *insmod ./upk-proc.ko*, следует перейти в директорию */proc* и убедиться в наличии каталога *upk_directory*, в котором должны находиться файлы *file1* и *file2*, как показано на рисунке 2.6.

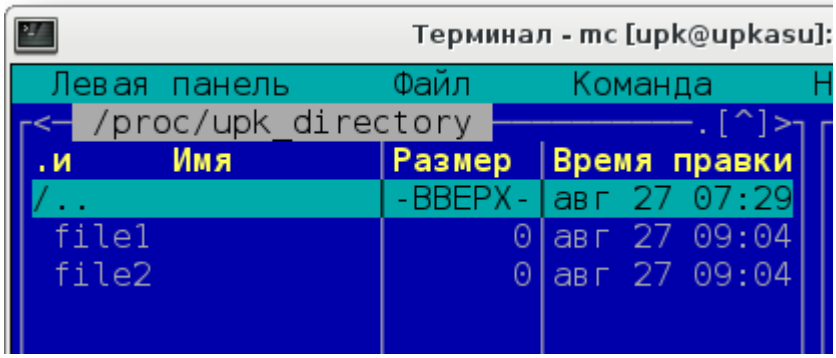

Рисунок 2.6 — Файлы, созданные модулем upk-proc.ko

Эти файлы не имеют начального содержания, в чем можно убедиться непосредственным их просмотром.

Чтобы изменить содержание созданных файлов, можно воспользоваться командой *echo*, например:

```
    echo "Новое содержание файла: file1" > /proc/upk_directory/file1
```

Измененное содержимое команды показано на рисунке 2.7.

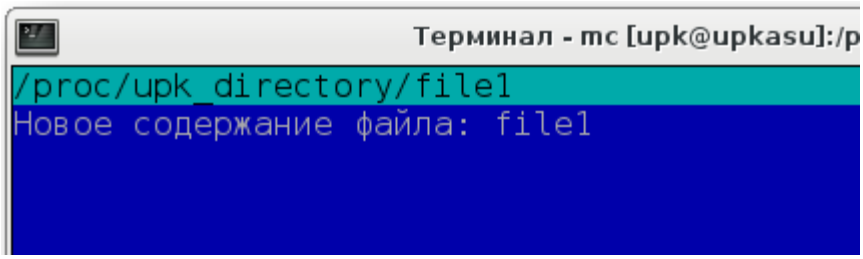

Рисунок 2.7 — Измененное содержимое файла /proc/upk_directory/file1

Замечание

Загрузка и выгрузка модуля регистрируются в файле системного журнала, в чем можно убедиться командой *dmesg* .
Кроме того, имена модуля регистрируются в файле */proc/kallsyms*, в чем можно убедиться командой:

```
    cat /proc/kallsyms | grep upk_proc
```

Результаты данного и предыдущего исследований следует отразить в личном отчете.

2.3 Модуль для работы с псевдофайловой системой sysfs

Хотя файловая система *sysfs* создана специально для ОС Linux, она наглядно показывает современные тенденции развития ядер всех ОС:

- стремление иметь универсальный текстовый интерфейс к информации ядра;
- освободиться от «непрозрачного» интерфейса, который обеспечивается классическими средствами управления устройствами ЭВМ, реализованными через системный вызов *ioctl()*.

Технология реализации *sysfs*, во многом, основана на идеях реализации файловой системы *procfs*, но ее основное назначение — обслуживание потребностей проекта *udev*.

Основная цель данного учебного исследования — практическое закрепление знаний о файловой системе *sysfs*, ограниченное использованием лишь одного технологического решения, основанного на идее классов.

Идея классов, как и в программировании, основана на понятии объектов, которые имеют *имена* и *именованные атрибуты* (свойства объекта):

- *имена объектов* отражаются в виде имен каталогов в директории */sys/class*;
- *именованные атрибуты* отображаются в виде имен текстовых файлов в директории */sys/class/<имя объекта>*;
- *содержание именованных атрибутов* — короткие строковые сообщения, которые могут читаться или записываться, в зависимости от прав доступа.

Эффективность использования идеи классов — значительное сокращение объема кода, необходимое программисту для реализации модуля, поддерживающего объект с большим количеством атрибутов.

Методическое обеспечение такого исследования предполагает, что студент:

- изучил теоретическую часть методического пособия, изложенную в подразделе 1.4;
- запустил ОС УПК АСУ и успешно подключил тему обучения;
- находится в системе, зарегистрировавшись пользователем *upk*;
- создал директорию *~/lab3_sos/3*, для размещения и сохранения результатов исследования.

Реализация проекта по созданию модуля состоит в следующем:

- в директории *~/lab3_sos/3*, создается текстовый файл *upk-class.c*, в который переносится содержимое листинга 1.5 (стр. 26);
- в директории *~/lab3_sos/3*, создается текстовый файл *Makefile*, в котором содержатся операторы, необходимые для создания модуля;
- с помощью утилиты *make*, от имени пользователя *root*, проводится создание модуля *upk-class.ko*.

Замечание

Значительное сокращение кода модуля обеспечивается следующими причинами:

- *отказом* от применения традиционных файловых операций, поддерживаемых в явном виде через структуру *file_operations*;
- *однотипностью* операций работы с атрибутами объекта, представленными только двумя действиями: чтением значения атрибута (*show*) и запись их значений (*store*);
- широким использованием макросов, возможность эффективного применения которых обусловлена первыми двумя причинами.

Технология исследования модуля *upk-class.ko* — аналогична предыдущему случаю:

- после загрузки модуля, в директории */sys/class* появляется каталог *upk-class*, в котором будут находиться три файла атрибутов, как показано на рисунке 2.8;
- файлы атрибутов имеют начальные значения, как показано на рисунке 2.9;
- изменить значение любого атрибута можно от имени пользователя root, например, командой:

```
echo "Всем привет\!\!\!" > /sys/class/upk-class/data3
```

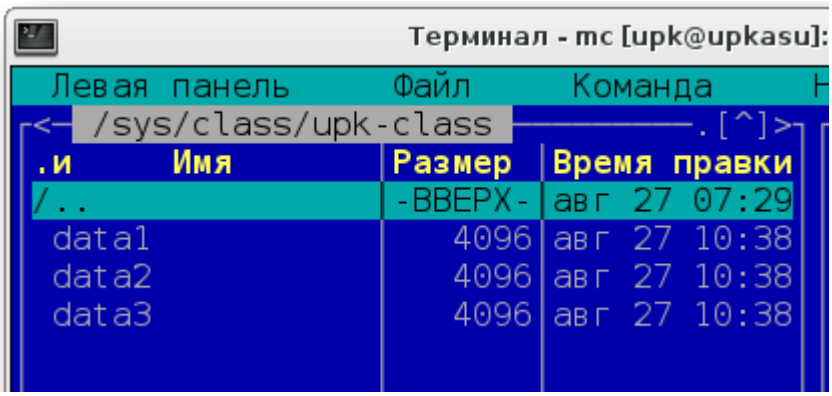

Рисунок 2.8 — Файлы атрибутов модуля upk-class.ko

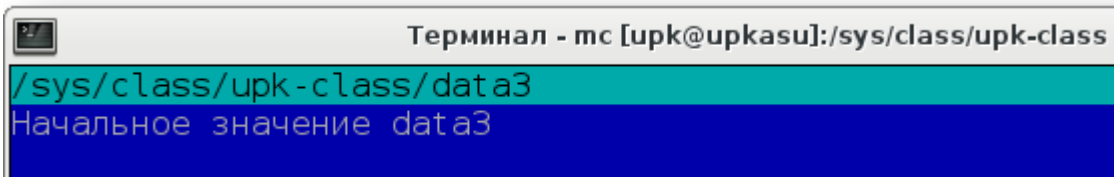

Рисунок 2.9 — Содержимое файла атрибута data3

По завершению исследования, следует отразить его результаты в личном отчете, а также подвести итоги работы по всей теме.

Список использованных источников

1 Резник В.Г. Операционные системы. Самостоятельная и индивидуальная работа студента. Учебно-методическое пособие. – Томск, ТУСУР, 2015. – 13 с.

2 Гордеев А.В. Операционные системы: учебное пособие для вузов. ─ СПб.: Питер, 2004. ─ 415с.

3 Таненбаум Э. Современные операционные системы. - СПб.: Питер, 2007. - 1037с.

4 Резник В.Г. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 38 с.

5 Резник В.Г. Современные операционные системы. Тема 2. Модульная структура ядра ОС. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 58 с.

Учебное издание

Резник Виталий Григорьевич

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебно-методическое пособие предназначено для изучения темы №3 по дисциплине «Современные операционные системы» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Учебно-методическое пособие

Усл. печ. л. . Тираж 100. Заказ .
Томский государственный университет
систем управления и радиоэлектроники
634050, г. Томск, пр. Ленина, 40