Московский государственный технический университет
имени Н. Э. Баумана

Учебное пособие

В.А. Крищенко, Н.Ю. Рязанова

ОСНОВЫ ПРОГРАММИРОВАНИЯ В ЯДРЕ ОПЕРАЦИОННОЙ СИСТЕМЫ GNU/LINUX

Издательство МГТУ им. Н. Э. Баумана

Московский государственный технический университет
имени Н.Э. Баумана

В.А. Крищенко, Н.Ю. Рязанова

ОСНОВЫ ПРОГРАММИРОВАНИЯ В ЯДРЕ ОПЕРАЦИОННОЙ СИСТЕМЫ GNU/LINUX

*Рекомендовано Научно-методическим советом
МГТУ им. Н.Э. Баумана
в качестве учебного пособия*

Москва
Издательство МГТУ им. Н.Э. Баумана
2010

УДК 681.326(075.8)
ББК 22.18
К82

Рецензенты: *Г.В. Зеленко, С.Б. Ткачев*

Крищенко В. А.

К82 Основы программирования в ядре операционной системы GNU/Linux : учеб. пособие / В.А. Крищенко, Н.Ю. Рязанова. – М. : Изд-во МГТУ им. Н. Э. Баумана, 2010. – 34, [2] с. : ил.

В пособии описаны основы создания программного кода, работающего в режиме ядра операционной системы GNU/Linux. Рассмотрены основы организации ядра Linux, создания подключаемых к ядру модулей, внесения изменений в исходный код ядра, его сборка и установка. Освещены вопросы синхронизации в ядре, выделения памяти и создания динамических структур данных, перехвата событий ядра, приемы отладки кода ядра, а также способы обмена данными между прикладными программами и ядром операционной системы.

Для студентов 3-го курса, специализирующихся по кафедре «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н.Э. Баумана.

УДК 681.326(075.8)
ББК 22.18

© МГТУ им. Н.Э. Баумана, 2010

ВВЕДЕНИЕ

Создание программного кода, работающего в ядре любой операционной системы, является специфической задачей.

В данном учебном пособии изложены основные сведения, необходимые для программирования в ядре операционной системы GNU/Linux. Предполагается, что читатель знает язык программирования Си [1] и имеет общее представление об интерпретаторе командной строки и основных служебных программах GNU[1], компиляторе GNU C и системе сборки GNU Make, знаком с основными понятиями из теории операционных систем [2] и unix-подобными операционными системами [3].

В пособии рассматривается операционная система GNU/Linux в вариантах Debian или Ubuntu. Примеры программ, приведенные в пособии, работоспособны для версии ядра Linux 2.6.26 и версии дистрибутива Debian GNU/Linux 5.0.

Для более детального изучения программирования в ядре можно использовать работы [4—8].

Несмотря на разнообразие посвященной ядру Linux литературы, единственным заведомо актуальным источником информации о ядре являются исходные коды используемой версии ядра в силу постоянного изменения внутренних структур данных ядра и заголовков функций.

Для ядра Linux практически не существует какой-либо программной документации, отличной от самих исходных кодов ядра с комментариями и содержимого каталога Documentation в архиве ядра. Для удобного поиска как мест определения, так и мест использования различных глобальных символов (функций, макросов,

[1] Документация по средствам разработки GNU: `http://www.gnu.org/manual`.

типов данных и глобальных переменных) может применяться специализированная поисковая система Cross Reference[2] по исходным кодам ядра Linux. Имена команд, файлов и каталогов, функций, а также фрагменты исходного кода выделены в тексте пособия моноширным шрифтом. Команды, вводимые пользователям, начинаются с символа-приглашения: $ make install

[2]Находится по адресу: `http://lxr.linux.no/linux`.

1. НАЧАЛО РАБОТЫ С ИСХОДНЫМ КОДОМ ЯДРА LINUX

1.1. Краткие сведения о ядре

Linux является ядром многопользовательской многозадачной операционной системы и поддерживает (с ограничениями) стандарты POSIX и Single UNIX Specification[3], а также ряд собственных программных интерфейсов.

Ядро Linux состоит из следующих основных компонентов: планировщик процессов; менеджер памяти; подсистема ввода-вывода и драйверы устройств; сетевая подсистема; виртуальная файловая система; подсистема межпроцессного взаимодействия и др. Отдельно следует упомянуть интерфейс системных вызовов ядра, обычно используемый прикладными программами через стандартную библиотеку языка Си, например GNU C Library.

С точки зрения программной организации ядро является монолитным. Часть кода ядра обычно находится в подключаемых модулях, которые могут загружаться и выгружаться динамически без перезагрузки системы. В виде модулей оформляются, например, драйверы устройств или код, ответственный за конкретные файловые системы. На этапе загрузки ядра необходимые модули могут загружаться с диска или из временного диска, организуемого с помощью начального загрузчика операционной системы.

1.2. Подготовка рабочего места программиста

В дальнейшем будет предполагаться, что работа с исходным кодом ядра ведется в каталоге, имя которого хранится в переменной окружения `$SYSPROG`, а рабочая версия файлов находится в

[3]Стандарты доступны по адресу: `http://www.opengroup.org`.

каталоге $SRC. Экспортируем переменные окружения и создадим рабочий каталог для работы с исходными кодом ядра:

```
$ export SYSPROG=~/sysprog
$ # Предполагается работа с ядром 2.6.26
$ export SRC=${SYSPROG}/linux-source-2.6.26
$ mkdir -p ${SYSPROG}
```

Для работы с исходными кодами ядра и сборки ядра требуется около трех гигабайт свободного дискового пространства. Для сборки ядра необходимы компилятор Си и средство сборки Make, которые устанавливаются через пакет `build-essential`. Для работы с меню конфигурации ядра следует поставить пакет библиотеки ncurses5, а для разархивирования (распаковки) исходных текстов – пакет программы сжатия Bzip2. Выполнить эти операции можно с помощью следующей команды:

```
$ sudo apt-get install build-essential \
    libncurses5-dev bzip2
```

Если предполагается создавать только модули для используемого в дистрибутиве ядра, то достаточно установить заголовочные файлы для этого ядра:

```
$ sudo apt-get install linux-headers-$(uname -r)
```

Если предполагается изучение исходных кодов ядра или их модификация, то следует поставить все дерево его исходных кодов. Ядро Linux разрабатывается с использованием системы управления версиями Git, поэтому удобно устанавливать ее с помощью такой команды:

```
$ sudo apt-get install git-core
```

Исходные коды следует взять из используемого дистрибутива, а затем создать локальный репозиторий системы Git. Для установки исходных кодов ядра дистрибутива следует установить пакет `linux-source` и затем развернуть полученный архив из каталога `/usr/src/linux` в желаемый каталог. Для этого выполняют следующие команды:

```
$ export VER=$(uname -r | cut -f1 -d-)
$ sudo apt-get install linux-source-$VER
$ cd $SYSPROG
$ tar xf /usr/src/linux-source-$VER.tar.bz2
```

Если необходимо внести изменения в какую-то конкретную версию ядра, то ее исходные коды нужно получить с сервера `http://kernel.org`.

После развертывания архива с исходными кодами следует инициализировать репозиторий системы Git, указать на необходимость отслеживания файлов на языках Си и ассемблер[4], а затем зафиксировать текущее состояние в системе управления версиями, выполнив команды:

```
$ cd $SRC
$ git init # Инициализация репозитория.
$ git add *.c *.h # Список файлов для фиксации.
$ git commit -m "Kernel sources" # Фиксация.
```

В дальнейшем разницу между изменениями и первоначальными исходными кодами можно просмотреть по команде `git diff`. Эту разницу принято называть патчем (от англ. *patch*).

Для написания исходного кода можно использовать как текстовые редакторы с подсветкой синтаксиса и интеграцией с системой сборки (Vim, Scite, Kate), так и специализированные среды разработки с поддержкой языка Си (KDevelop, Anjuta, Eclipse и даже MS VS с компиляцией через ssh). В качестве базового варианта для работы с ядром или модулем ядра можно предложить среду KDevelop. Для поиска объявлений типов и функций можно использовать программу `ctags`, которая интегрирована со средой KDevelop. Установка KDevelop и ctags осуществляется в Debian следующей командой:

```
$ sudo apt-get install kdevelop exuberant-ctags
```

При использовании KDevelop исходный код ядра можно импортировать как проект на языке Си со своим собственным файлом сборки. В проект нет необходимости добавлять все файлы ядра (это очень ресурсоемкая операция), достаточно добавить основную их часть или вообще не добавлять ничего, поскольку это не мешает проиндексировать все файлы кода ядра служебной программой `ctags`, интерфейс к которой доступен прямо из KDevelop. Это позволит быстро перемещаться к объявлениям и определениям

[4]При изменении прочих файлов список отслеживаемых системой контроля версий файлов следует расширить.

функций, макросов и типов данных через контекстное меню редактора среды или с помощью поля поиска символов.

1.3. Обзор исходных текстов ядра

Ядро Linux поддерживает в настоящее время более двух десятков архитектур ЭВМ, как 32-, так и 64-разрядных, с различным порядком байт в слове. Весь код ядра можно разделить на архитектурно-зависимый (каталог `arch`) и архитектурно-независимый, который не включает какого-либо ассемблерного кода и написан исключительно на языке программирования Си.

В каталоге `include` находятся заголовочные файлы. Зависящая от аппаратной архитектуры часть заголовочных файлов вынесена в каталоги вида `include/asm-$ARCH/`. При настройке ядра создается ссылка `include/asm` на каталог файлов для архитектуры используемой ЭВМ. Заголовочные файлы ядра, расположенные в каталоге `inlude/linux/`, включают в себя архитектурно-зависимые заголовочные файлы.

Благодаря этому в основных исходных кодах ядра (файлах на языке Си) в директивах `#include` не используются ссылки на каталоги `asm` или `arch`. В заголовочных файлах ядра наибольший интерес представляет собой содержимое каталога `include/linux`, предназначенного для включения в модули ядра.

Следует отметить, что с аппаратной архитектурой связаны многочисленные параметры в конфигурации ядра. Например, за архитектуру x86 отвечают три основных параметра: `CONFIG_X86`, `CONFIG_X86_32`, `CONFIG_X86_64` и множество дополнительных, например параметр `CONFIG_X86_SMP`, отвечающий за поддержку многопроцессорной архитектуры. Таким образом, аппаратные архитектуры i386 и amd64 в современных версиях ядра объединены в одну архитектуру[5] x86. При необходимости использовать различающийся код для каждой из них применяются директивы условной компиляции, как показано ниже:

```
# ifdef CONFIG_X86_32
#  include "unistd_32.h"
```

[5] До ядра версии 2.6.23 архитектуры i368 и amd64 рассматривались как принципиально различные.

```
# endif
# ifdef CONFIG_X86_64
#  include "unistd_64.h"
# endif
```

Исходные коды ядра разделены по своему назначению на несколько каталогов. Следующие из них представляют наибольший интерес:

`kernel` – основные фукции ядра, планировщик задач;

`mm` – управление оперативной памятью;

`ipc` – межпроцессное взаимодействие;

`lib` – библиотека вспомогательных функций;

`fs` – поддержка файловых систем, виртуальная файловая система;

`net` – реализация сетевых протоколов и фильтрации пакетов;

`init` – начальная загрузка ядра.

1.4. Сборка и установка ядра

При системном программировании типичной операцией является сборка ядра из исходных кодов. Данная операция может понадобиться в следующих случаях:

- при внесении изменений в исходный код ядра;
- при изменении конфигурации ядра.

Обычно часть модулей ядра включается в загрузочный вре́менный диск initrd (initial ramdisk), и тогда для загрузки нового ядра требуются средства создания образа initrd. Для их установки можно поставить следующий пакет:

```
$ sudo apt-get install initramfs-tools
```

Процесс сборки ядра из исходных кодов управляется конфигурацией ядра, которая хранится в файле `$SRC/.config`. В качестве начальной конфигурации можно взять, например, используемую конфигурацию ядра:

```
$ cp /boot/config-$(uname -r)  $SRC/.config
```

Установка ядра является процессом, затрагивающим множество файлов в операционной системе. В силу этого в большинстве случаев следует применять зависящий от дистрибутива подход, основанный на использовании пакетного менеджера. Поэтому сначала будет рассмотрен общий подход, а затем подход дистрибутива Debian.

Сборка и установка ядра: общий подход

Для идентификации создаваемой версии ядра следует изменить его версию, заданную в файле `$SRC/Makefile`. Рекомендуется поменять значения поля EXTAVERSION на любое другое, например на «-test».

Присвоим переменной VER номер полной версии нового ядра:

```
$ export VERSION=2.6.26-test
```

В файле сборки ядра заданы несколько целей, позволяющих проводить типичные операции с ядром с помощью команды `make`, отданной в каталоге `$SRC`.

Если конфигурация ядра была взята от другой версии, то следует выполнить преобразование конфигурации командой `make oldconfig`.

Для изменения конфигурации ядра обычно используется команда `make menuconfig`. С помощью этой команды можно создать компактное и быстро собирающееся ядро, но делать это следует с большой осторожностью. Например, можно отключить поддержку отсутствующего оборудования на данной машине, неиспользуемых сетевых протоколов и файловых систем.

После изменения конфигурации ядра или исходного кода ядра его следует собрать заново командой `make`. Команда `make modules_install` устанавливает скомпилированные модули ядра и систему сборки новых модулей в каталог `libmodules`. Для установки самого ядра в каталог `/boot` применяется команда `make install`.

Для загрузки ядра требуется, помимо установки ядра, создать образ вре́менного загрузочного диска initrd, который считывается загрузчиком операционной системы, временно монтируется в качестве корневой файловой системы и затем используется для загрузки необходимых модулей до монтирования корня файловой системы на раздел жесткого диска. Этот механизм позволяет оформлять в виде модулей драйверы дисковых контроллеров и файловых систем.

Следует отметить, что при небольших изменениях в ядре дистрибутива можно просто взять его образ, но в общем случае для создания образа диска используется специальная программа, например, `mkinitramfs` из пакета initramfs-tools. Следующие команды

создают образ диска initrd и копируют его в каталог начальной загрузки:

```
$ cd ${SRC}
$ /usr/sbin/mkinitramfs -o \
      initrd.img-$VER $VER
$ sudo cp initrd.img-$VER /boot
```

После установки ядра и образа загрузочного диска следует создать в файле загрузчика раздел для загрузки нового ядра с использованием созданного образа[6]. Это можно сделать по образцу уже имеющихся записей об установленных ядрах. Ни в коем случае не следует изменять существующие записи для загрузки нового ядра, поскольку новое ядро может работать некорректно.

Сборка и установка ядра: подход операционной системы Debian

В случае дистрибутива Debian изменения в файле `/boot/grub/menu.lst` и установку ядра лучше производить не вручную, а рекомендованным в дистрибутиве способом, который позволяет создать два пакета: один с ядром и диском initrd, второй – с заголовками для сборки модулей. Для этого необходимо установить два следующих пакета:

```
$ sudo apt-get install kernel-package fakeroot
```

Изменения конфигурации ядра при этом происходят описанным в предыдущем разделе способом, а сборка ядра и создание пакетов с ядром и заголовочными файлами ядра осуществляются особым образом. Сначала следует инициализировать систему создания пакетов. Для этого выполняют следующую команду:

```
$ cd $SRC && make-kpkg clean
```

Для сборки нужно выполнить такую команду:

```
$ fakeroot make-kpkg --append-to-version=-test \
          --initrd kernel_image kernel_headers
```

Здесь -test – подверсия ядра; таким образом, редактировать файл Makefile не требуется; в случае удачной сборки в каталоге `$SYSPROG` появятся два пакета файла с расширением `.deb`;

[6] В случае GNU Grub изменить следует файл `/boot/grub/menu.lst`.

параметр `kernel_headers` можно не использовать, если не предполагается разработка модулей для данного ядра.

В случае выявления ошибок при сборке эту команду повторяют многократно после их устранения. Данная команда вызывает команду `make` в ходе своей работы.

Установка пакета ядра приводит к копированию образов ядра и виртуального диска в каталог `/boot`, а также к установке модулей и редактированию меню загрузки. Установка пакета заголовочных файлов аналогична выполнению команды `make modules_install`. Созданные пакеты можно установить с помощью стандартной утилиты управления пакетами `dpkg`, как показано ниже:

```
$ cd $SYSPROG && sudo dpkg -i linux*.deb
```

Для удаления созданных пакетов используют программу управления пакетами `apt-get`. Следующая команда удалит все пакеты ядра с подверсией test:

```
$ sudo apt-get remove linux.*test
```

Преимуществом описанного подхода является возможность полного и корректного восстановления состояния системы путем удаления пакетов.

2. ОСНОВЫ ПРОГРАММИРОВАНИЯ ДЛЯ ЯДРА LINUX

2.1. Оформление исходного кода ядра

Исходные коды ядра операционной системы GNU/Linux состоят главным образом из кода на языке Си диалекта GNU C. Остальная часть кода ядра представлена кодом для транслятора GNU Assembler для различных поддерживаемых ядром аппаратных архитектур. В стандарте кодирования ядра не используются никакие специфичные для языка Си++ расширения, включая даже однострочные комментарии[7].

Для проверки соблюдения принятого в ядре стандарта кодирования существует входящая в архив исходных кодов ядра программа `checkpatch.pl`, которая может проверять как патчи, так и

[7] В ядре могут присутствовать файлы, нарушающие эти требования.

отдельные файлы (ключ `file`). Эта программа находится в каталоге `$SRC/scripts`.

Документ, описывающий стандарты кодирования, прилагается к исходным кодам ядра (`Documentation/CodingStyle`). Этот документ должен быть изучен любым программистом, собирающимся писать патч к ядру или модуль ядра, и здесь не приводится. Отметим, что для оформления отступов в исходных кодах ядра применяется только табуляция с предположением, что она представлена в редакторах кода восемью пробелами[8].

Поскольку в языке Си нет исключений, в коде ядра для индикации ошибки используется результат функции в соответствии со следующим соглашением:

- если функция возвращает некоторый указатель, то его ненулевое значение свидетельствует о нормальном выполнении, а нулевое – об ошибке;
- остальные функции возвращают некоторое целое значение, нулевое значение означает отсутвие ошибки, ненулевое – некоторый код ошибки.

Так как в языке Си нет конструкций обработки исключений, вместо них используется оператор безусловного перехода на метки, расположенный в конце функции. Данный оператор может применяться и для выхода из вложенного цикла. Иное использование данного оператора в исходном коде ядра не рекомендуется. Типичный случай применения оператора перехода при выделении ресурсов выглядит следущим образом:

```
int do_work()
{
    int result = ERROR;
    other_resource or;
    some_resource *sr = create_some_resource();
    if (sr == NULL)
        goto out;
    if (create_other_resource(&or) != 0) {
        result = OTHER_ERROR;
        goto out1;
```

[8] В данном пособии табуляция представлена четырьмя пробелами из-за меньшей по сравнению с экраном ширины печатного листа.

```
    }
    if (!do_something(sr, &or))
        result = 0; /* успешное завершение */
    free_other_resource(&or);
out1:
    free_some_resource(sr);
out:
    return result;
}
```

Исходный код ядра следует по возможности компилировать с флагами `-Wall -Werr`. Правильно оформленный код не должен генерировать каких-либо предупреждений при компиляции.

2.2. Внесение изменений в исходный код ядра

При внесении изменений в основной код ядра необходима его сборка из полных исходных кодов и новая загрузка системы, в то время как для создания модуля ядра достаточно наличия заголовочных файлов ядра и системы сборки модулей, а загрузка и выгрузка модулей может проводиться «на лету». Разработка модулей гораздо более удобна, но далеко не все желаемые изменения можно выполнить, используя модуль. Например, установить перехватчик каких-либо событий ядра невозможно без внесения некоторых изменений в его исходный код.

Стандартной формой для представления изменений в исходные тексты ядра является патч, который отражает разницу между исходным и измененным набором исходных кодов ядра. Патчи создаются, например, с помощью программы `diff` в соответствии с документацией в файле `Documentation/patches`. Патчи применяются к исходным кодам с помощью программы `patch`:

```
$ cd ${SYSPROG}
$ # Создание патча сразу для всех изменений
$ diff -uprN -X  ${SRC}/Documentation/dontdiff \
    ${SRC}.orig/ ${SRC}/ > some_patch
```

Полученный патч представляет собою простой текстовый файл, при необходимости его можно открыть и определить в текстовом редакторе. Для проверки соответствия патча стандар-

там кодирования, принятым в ядре Linux, служит программа `scripts/checkpatch.pl`.

При использовании системы управления версиями системы Git получить патч можно с ее помощью следующим образом:

```
$ cd $SRC && git diff > ../some_patch
$ scripts/checkpatch.pl ../some_patch #Проверка.
```

Для применения патча следует перейти в каталог с исходными кодами, которые необходимо изменить, и выполнить, например, следующую команду:

```
$ patch -p1 < $SYSPROG/some_patch
```

2.3. Создание модулей ядра

Модуль ядра представляет собой объектный файл, загружаемый в адресное пространство ядра и динамически компонуемый (линкуемый) с ядром. Код модуля ядра выполняется с теми же привилегиями, что и остальная часть кода ядра. Основное ограничение модулей заключается в том, что они не могут использовать функции ядра, которые не экспортируются.

Модуль ядра может загружаться в момент загрузки ядра из вре́менного диска начальной загрузки или из каталога `/lib/modules/$(uname -r)` после монтирования корневого каталога.

Для операций с модулями существует несколько специальных служебных программ.

Для добавления и удаления постоянно загружаемых модулей используется программа `modprobe`. Модуль может подгружаться и выгружаться в ходе работы системы программами `insmod` и `rmmod`. Произведенные этими программами изменения не сохраняются после перезагрузки ядра. Служебная программа `lsmod` отображает список загруженных модулей.

С точки зрения программиста исходный код простого модуля ядра представляет собой несколько исходных файлов для компилятора GNU C и файл для системы сборки GNU Make. Для сборки модуля ядра достаточно иметь установленные заголовочные файлы ядра и систему сборки модулей ядра.

Создание объектного файла модуля ядра значительно отличается от создания объектных файлов пользовательских программ,

поэтому для упрощения этой задачи служит система сборки модулей ядра. Для ее применения следует вызвать make-файл системы сборки модулей из файла сборки конкретного модуля. Система сборки модулей находится в каталоге `/lib/modules/<версия ядра>/build`. Она появляется там как после установки пакета с заголовками ядра (для дистрибутивного ядра), так и после выполнения команды `make modules_install` (в случае нового ядра). Модуль должен собираться с помощью системы сборки именно того же ядра, с которым он затем будет работать, иначе попытка загрузить модель приведет к ошибке (при загрузке модуля проверяются полные версии ядер).

Исходный код модуля ядра обычно включает несколько файлов на языке Си и файл `Makefile` для системы сборки GNU Make. Последний может быть организован несколькими способами, далее рассматривается один из них:

```
# Используется текущее ядро:
KERNELVERSION = $(shell uname -r)
# Использовать другое ядро:
# KERNELVERSION = 2.6.26-test
# Каталог системы сборки модулей ядра:
KERNELDIR = /lib/modules/$(KERNELVERSION)/build
# Каталог, где лежат исходники модуля:
PWD = $(shell pwd)
# Строгая проверка предупреждений gcc:
EXTRA_CFLAGS = -Wall -Werror
ifneq ($(KERNELRELEASE),)
# Если это сборка ядра, то
# перечислить собираемые модули ядра:
    obj-m := test_module.o
else
# Если это еще не сборка ядра, то объявить целью
# по умолчанию вызов системы сборки ядра:
default:
    $(MAKE) -C $(KERNELDIR) M="$(PWD)" modules
endif
```

Как видно из комментариев, файл `Makefile` сначала вызывает систему сборки модуля ядра в цели `default`. Затем этот же файл будет включен системой сборки модулей (переменная

KERNELRELEASE при этом уже будет установлена) и должен сообщить ей имя модуля. В данном простейшем случае модулю ядра будет соответствовать единственный файл исходного кода с тем же именем (test_module.c).

Две функции модуля ядра являются выделенными: функция инициализации и функция выхода. После загрузки модуля и его связывания с остальной частью кода ядро вызывает функцию инициализации модуля. Перед выгрузкой модуля вызывается функция выхода модуля. Одним из назначений функции инициализации является установка точек входа в модуль, например регистрация его в качестве драйвера устройства или регистрация функций модуля в качестве перехватчиков какой-либо активности ядра.

Существует несколько способов указания этих функций. Наиболее типичный – использование макросов module_init и module_exit. Они добавят функции инициализации init_module и cleanup_module. В ходе работы системы сборки специальная служебная программа modpost проанализирует имена в объектном коде модуля и создаст файл метаданных модуля с расширением .mod.c, который затем используется при создании объектного файла модуля. Содержимое файла метаданных рекомендуется для самостоятельного изучения, так же, как и применение программы readelf к полученным объектным файлам.

3. ПРОГРАММНЫЕ ИНТЕРФЕЙСЫ ЯДРА LINUX

3.1. Служебные функции ядра

Одна из особенностей программирования в режиме ядра – принципиальная невозможность использовать какие-либо программные библиотеки, включая стандартную библиотеку языка Си. Все доступные системному программисту функции должны находиться в ядре, поэтому для облегчения его работы в ядро включены многочисленные сервисные функции. Основными из них являются следующие:

1) функции для обработки строк и областей памяти, аналогичные имеющимся в стандартной библиотеке языка Си. Их прототипы доступны через заголовочный файл linux/string.h;

2) функции sprintf, sscanf и им подобные, а также функции отладочной печати, доступные в файле `linux/kernel.h`;

3) функции динамического выделения памяти, прототипы которых находятся в файле `linux/slab.h`;

4) функции и макросы для создания списков, доступные через файлы `linux/lists.h` и `linux/plists.h`;

5) функции организации синхронизации и работы с нитями (threads) доступные после включения файла `linux/sched.h`;

6) функции и макросы поддержки модулей, доступные в файле `<linux/module.h>`.

Следует отметить, что заголовочные файлы ядра включены друг в друга, в результате прототипы одной и той же функции могут быть доступны после включения различных файлов.

Ряд вспомогательных функций соответствует стандартным названиям и назначениям (например, `strlen`) или имеют букву «k» в названии (kernel), например `kmalloc`, но многие сервисные функции не имеют аналогов в стандартной библиотеке, например `memparse`.

Функции логирования (`printk` и другие из файла `linux/kernel.h`) передают информацию службе `klogd` и являются важным инструментом отладки внутриядерного кода. Использование функции `printk` выглядит примерно следующим образом:

```
printk(KERN_INFO "Period = %d\n", period);
```

Находящийся перед форматной строкой макрос указывает на тип сообщения (ошибка, предупреждение, информация). Для просмотра сообщений ядра можно использовать команды `dmesg` или `cat /proc/kmsg`.

3.2. Выделение памяти и связанные списки

Выделение и освобождение динамической памяти в ядре несколько отличается от привычного для прикладных программ на языке Си. Для выделения памяти обычно используются функции `kmalloc`, `kcalloc`, `kzalloc`, для освобождения – функция kfree.

Кроме того, программист имеет доступ к интерфейсу выделения памяти через ряд функций с префиксом `kmemcache`. Использование этих функций рекомендуется в случае частого выделения

и освобождения одинаковых блоков памяти одинакового размера с одинаковым начальным состоянием. Реализация функций типа `kmalloc` также осуществляется через данный механизм кэша.

Кроме требуемого объема памяти, все функции выделения памяти в ядре принимают параметр, определяющий необходимое поведение ядра при выделении памяти и свойства выделенного блока. Существует множество флагов, обусловливающих поведение аллокатора, полностью они описаны в файле `include/linux/gfp.h`. Наиболее часто используются два флага: `GFP_ATOMIC` и `GFP_KERNEL`.

Флаг `GFP_ATOMIC` означает, что при выделении памяти не произойдет смена контекста. Этот флаг используется, например, при обработке прерываний, для выделения небольших фрагментов памяти. Если необходимый объем памяти не может быть выделен сразу же, то функция вернет нулевой указатель.

Флаг `GFP_KERNEL` служит для выделения значительных объемов памяти в некритических случаях. При его использовании вызывающая нить может быть заблокирована, если это необходимо для выделения памяти. Например, дисковый кэш может быть сброшен на диск для высвобождения памяти или данные какого-либо процесса могут быть помещены в область подкачки. Если же и подобные операции не приводят к освобождению требуемого объема памяти, функция вернет нулевой указатель.

Ядро Linux предлагает механизмы для организации связанных списков (файл `list.h`) и связанных списков с приоритетами (файл `plist.h`), причем создание механизмов с аналогичной функциональностью программистом крайне нежелательно. В списках ядра вместо часто предлагаемой в учебниках весьма неэффективной структуры, которая содержит указатель на данные и на соседние элементы списка, используется непосредственное добавление структуры типа `list_head` в качестве поля в структуру, содержащую данные.

Реализация списков в ядре позволяет выделять для каждого элемента списка только один блок памяти, что значительно сокращает накладные расходы на организацию списков и дает набор функций и макросов для манипулирования любыми списками.

Для хранения «головы» и «хвоста» списка применяется отдельная структура, для ее объявления и инициализации можно использовать макрос LIST_HEAD.

Нижеследующая функция создает список из пяти элементов с помощью функции list_add и демонстрирует его обход с помощью макроса list_for_each, для удаления элемента из списка применяется функция list_del:

```
#include <linux/slab.h>
#include <linux/list.h>
struct data {
    int n;
    struct list_head list;
};
void test_lists(void)
{
    struct list_head *itr, *itr_safe;
    struct data *item;
    int i;
    /* Макрос, объявляющий struct list_head list
       и инициализирующий ее */
    LIST_HEAD(list);
    for (i = 0; i < 10; i++) {
      item = kmalloc(sizeof(*item), GFP_KERNEL);
        if (!item) goto out;
        item->n = i;
        list_add(&(item->list), &list);
    }
    list_for_each(itr, &list) {
      item = list_entry(itr, struct data, list);
       printk(KERN_INFO "[LIST] %d\n", item->n);
    }
out:
    list_for_each_safe(itr, itr_safe, &list) {
      item = list_entry(itr, struct data, list);
        list_del(itr);
        kfree(item);
        }
}
```

Обход списка с целью удаления осуществляется с помощью макроса `list_for_each_safe`, который использует дополнительную переменную для безопасного удаления элементов списка при его обходе.

Макрос `list_entry` позволяет получить адрес структуры по адресу ее поля. Он эквивалентен макросу `container_of`, описанному в файле `linux/kernel.h` следующим образом:

```
#define container_of(ptr, type, member) ({ \
 typeof(((type *)0)->member) *mptr = (ptr); \
(type *)((char *)mptr - offsetof(type, member));\
})
```

Макрос `offsetof` описан в файле `include/linux/stddef.h`. С его помощью можно осуществить смещение заданного поля от начала структуры данного типа. Он реализуется либо с поддержкой компилятора, либо следующим образом:

```
#define offsetof(TYPE, MEMBER) \
    ((size_t) &((TYPE *)0)->MEMBER)
```

Данные макросы применяются в ядре не только для реализации списков, но и в ряде других случаев.

3.3. Системные вызовы

Системные вызовы – программный интерфейс ядра, предназначенный исключительно для использования процессами пользователя через некоторый архитектурно-зависимый интерфейс, обычно применяющий прерывания.

Для добавления в ядро нового системного вызова необходимо пересобрать ядро, предварительно добавив информацию о новом вызове в таблицы системных вызовов. Расположение и вид таблиц системных вызовов зависят от архитектуры системы. В случае архитектуры x86 таблица располагается в файле syscall_table_32.S.

Механизм системных вызовов определяется аппаратной архитектурой, поэтому для упрощения применения нового системного вызова существуют системный вызов `sys_syscall` и вызывающая его функция `syscall` в стандартной библиотеке. Эта функция позволяет сделать системный вызов по номеру этого вызова.

Таблица системных вызовов в текущих версиях ядра Linux не экспортируется и, следовательно, недоступна для модулей ядра.

Для решения это проблемы «силой» к ядру иногда применяют патч, экспортирующий ее и позволяющий таким образом установить указатель на свою функцию в таблицу системных вызовов. Однако делать так не следует, поскольку таблица системных вызовов не должна меняться после сборки ядра. Для решения задачи перехвата вызовов лучше создать отдельный механизм уведомления, как будет показано в разд. 6.

3.4. Обмен данными с прикладными программами

Обмен данными между ядром и пользовательскими программами осуществляется с помощью системных вызовов. Прикладным программам предоставлена возможность использовать системные вызовы работы с файлами для реализации информационного обмена с ядром. Для этого в GNU/Linux используется файловая система ProcFS, связанная с каталогом `/proc`. Данная файловая система позволяет установить callbacks-функции, вызываемые при чтении или записи данных пользователем в файлы в каталоге `/proc`.

Данные через ProcFS следует передавать в текстовом виде, используя только кодировку ASCII. Типичный вид данных – файл с одинаковыми по структуре строками, отдельные поля в которых разделены пробелами или табуляциями. Такой файл легко обрабатывать с помощью стандартных служебных программ `grep`, `cut`, `wc`, `awk` и др.

Для работы с ProcFS доступны два интерфейса: основной, описанный в файле `proc_fs.h`, и более новый и удобный в ряде случаев интерфейс последовательностей (sequence), описанный в файле `seq_file.h`. Достаточно подробно эти интерфейсы рассмотрены в [6].

Альтернативой применению ProcFS могут быть использование фиктивного символьного устройства и обмен данными через его файл в каталоге `/dev`. Реализация этого варианта позволит разобраться с написанием драйвера фиктивного устройства. Вопрос создания такого драйвера подробно описан в [4].

4. НИТИ И СИНХРОНИЗАЦИЯ В ЯДРЕ LINUX

4.1. Нити внутри ядра

Ядро Linux поддерживает симметричную многопроцессорную архитектуру(SMP) и внутриядерные нити (kernel threads). Для создания нити используется функция `kernel_thread`, доступная после директивы `#include <linux/schead.h>` и описанная в файлах `processor.h`.

Выполнение нити может быть прервано планировщиком задач, который передает управление другой нити. Поэтому для ликвидации проблемы «гонок» (race conditions) доступ к любым программным или аппаратным ресурсам, который теоретически может осуществляться более чем одной нитью, должен быть упорядочен (синхронизирован) явным образом. В частности, это касается изменения глобальных данных.

Для решения задачи синхронизации в ядре Linux существуют следующие механизмы синхронизации, которые обычно доступны после включения файла `linux/sched.h`,:

- переменные, локальные для каждого процессора (per-CPU variables), их интерфейс описан в файле `linux/percpu.h`;
- семафоры (`linux/semaphore.h`) и спин-блокировки `linux/spinlock.h`;
- семафоры читателей и писателей (`linux/rwsem.h`);
- мьютексы реального времени (`linux/rtmutex.h`);
- механизмы ожидания выполнения (см. `linux/completion.h`);
- атомарные переменные (описаны в архитектурно-зависимых файлах `atomic*.h`).

Некоторые из этих механизмов будут рассмотрены далее.

4.2. Механизм ожидания завершения

Достаточно часто встречающимся сценарием является запуск некоторой задачи в отдельной нити и ожидание завершения ее выполнения. В ядре нет аналога функции ожидания завершения нити, вместо нее требуется явно использовать механизмы синхронизации.

Для задачи ожидания какого-либо события не следует применять обычный семафор. В частности, реализация семафора оптимизирована исходя из предположения, что обычно семафор открыт. Поскольку в данном случае не идет речь о критических секциях, то для ожидания завершения лучше использовать не семафоры, а специальный механизм ожидания выполнения (completion). Этот механизм позволяет одному или нескольким нитям дожидаться наступления какого-то события, например, завершения другой нити или перехода ее в состояние готовности выполнять работу.

Следующий пример демонстрирует запуск нити и ожидание завершения его выполнения:

```
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/delay.h>
static int thread(void * data)
{
    struct completion *finished =
            (struct completion *)data;
    /* current -- указатель на дескриптор
       текущей задачи */
    printk(KERN_INFO "Thread [%p] is running\n",
           current);
    msleep(30000); /* Спать 3 с. */
    printk(KERN_INFO "Thread [%p] completed\n",
           current);
    /* Отмечаем факт выполнения условия. */
    complete(finished);
    return 0;
}
int test_thread(void)
{
    pid_t pid;
    DECLARE_COMPLETION(finished);
    /* Запускаем новую нить */
    pid = kernel_thread(thread, &finished,
        CLONE_FS);
    /* Дожидаемся выполнения условия. */
    wait_for_completion(&finished);
```

```
    printk(KERN_INFO "Thread [%p] completed\n",
           current);
    /* Имитируем неудачную загрузку модуля */
    return -1;
}
module_init(test_thread);
```

4.3. Семафоры, мьютексы и спин-блокировки

Семафор в ядре представляет собой механизм, позволяющий ограничить число нитей, которое одновременно выполняются в некоторой области кода. Это число называется значением семафора. Мьютекс (mutual exclusion) — это семафор, имеющий значение, равное единице. Для обозначения открытия и закрытия семафоров в ядре используются термины down и up соответственно.

Семафоры в ядре подобны POSIX-семафорам, используемым прикладными программами, но имеют несколько другой интерфейс. Существует очевидное ограничение на применение семафоров в ядре: их невозможно использовать в том коде, который не должен «уснуть», например при начальной обработке прерываний. Для синхронизации в последнем случае служит спин-блокировка (spinlock), использующая простое ожидание в цикле.

Неудачная попытка входа в критическую секцию с помощью семафоров означает перевод нити в «спящее» состояние и переключение контекста, что является дорогостоящей операцией. Поэтому если необходимость синхронизации связана только с наличием в системе нескольких процессоров, то для небольших критических секций следует применять спин-блокировку.

Кроме обычных мьютексов и семафоров в ядре существует новый интерфейс для мьютексов реального времени (rt mutex).

Особый, но часто встречающийся случай синхронизации — появление так называемых «читателей» и «писателей». «Читатели» только читают состояние некоторого ресурса и поэтому могут осуществлять к нему параллельный доступ. «Писатели» изменяют состояние ресурса и в силу этого должны иметь к ресурсу монопольный доступ, причем чтение ресурса в этот момент времени также должно быть заблокировано.

Для реализации «писателей» и «читателей» в ядре Linux существуют специальные версии семафоров и спин-блокировок. Мью-

тексы реального времени не имеют реализации для «читателей» и «писателей», поэтому разработчик должен сделать осознанный выбор механизма синхронизации.

Ниже приведен пример использования семафоров «читателей» и «писателей» при добавлении и поиске элемента в списке:

```
struct data {
    int value;
    struct list_head list;
};
static struct list_head list;
static struct rw_semaphore rw_sem;
int add_value(int value)
{
    struct data *item;
    item = kmalloc(sizeof(*item), GFP_ATOMIC);
    if (!item) goto out;
    item->value = value;
    down_write(&rw_sem);
    list_add(&(item->list), &list);
    up_write(&rw_sem);
    return 0;
out:
    return -ENOMEM;
}
int is_value(int value)
{
    int result = 0;
    struct data *item;
    struct list_head *itr;
    down_read(&rw_sem);
    list_for_each(itr, &list) {
      item = list_entry(itr, struct data, list);
        if (item->value == value) {
            result =  1; goto out;
        }
    }
out:
    up_read(&rw_sem);
```

```
    return result;
}
void init_list(void)
{
    init_rwsem(&rw_sem);
    INIT_LIST_HEAD(&list);
}
```

4.4. Использование атомарных переменных

Если разделяемый между нитями ресурс представляет собой единственную целочисленную глобальную переменную, то использование семафоров является избыточным. Однако отсутствие синхронизации приводит к тому, что даже результат выражения `a++` с глобальной переменной на некоторых архитектурах может приводить к неожиданному результату в случае параллельно выполняющихся нитей.

Для того чтобы гарантировать атомарность операций с целочисленными глобальными переменными, можно использовать особые атомарные переменные. Реализация последних зависит от архитектуры и может основываться на атомарных процессорных операциях с целыми числами или на спин-блокировках.

Интерфейс для использования атомарных переменных для архитектуры x86 можно увидеть в файлах `atomic_32.h` и `atomic_64.h` в каталоге `include/asm/x86`.

5. ОТЛАДКА ВНУТРИЯДЕРНОГО КОДА

Одна из очевидных трудностей программирования в режиме ядра — невозможность использования отладчика прикладных программ. Для решения этой проблемы можно применять специальный отладчик ядра, для чего, в частности, требуется пересобрать ядро и использовать отладку в виртуальной машине. Однако с помощью отладчика невозможно решить все проблемы создания надежных программ, поэтому ниже будут рассмотрены вопросы отладочной печати, проверки условий и модульного тестирования.

Важными методами отладки программ являются отладочная печать (ведение журнала работы) и проверка условий (конструкция assert). Для отладочной печати используют функцию `printk`, для удаления малозначимых сообщений из рабочей версии — директивы препроцессора, как показано ниже:

```
#ifdef DEBUG
#define log(format, ...) \
    printk(KERN_NOTICE  format, ## __VA_ARGS__)
#else
#define log(format, ...)
#endif
```

Отладочные сообщения должны ограничиться символами таблицы ASCII (в частности, недопустимо использовать русские буквы в любой кодировке). Для разбора правильно построенных отладочных сообщений удобно применять программы-фильтры `grep`, `tail` и им подобные. В ходе тестирования модуля сообщения ядра можно читать из файла `/proc/kmsg`.

В ядре нет стандартного макроса проверки условий `assert`, его аналог рекомендуется создать самостоятельно, например, следующим образом:

```
#define assert(expr) if (!(expr)) { \
  printk("Assertion (%s), %s, %s, line %d\n",\
     #expr, __FILE__, __FUNCTION__, __LINE__); \
  BUG(); \
}
void test_assert(void)
{
    assert(2 * 2 == 5);
}
```

При организации модульного тестирования (unit testing) разработчик может столкнуться с проблемой оформления тестов, так как создаваемый код обычно не может быть скомпилирован для работы в пользовательском режиме. Модульное тестирование части кода можно осуществить с помощью модуля примерно со следующей функцией инициализации:

```
int init(void)
{
```

```
    int i;
    for (i = 0; i< 3; i++) {
        test_001();
        test_002(); /* и т.д. */
        printk(KERN_INFO "Run %d completed., i);
    }
    return -1;
}
```

При загрузке этого модуля в ядро несколько раз выполняются тесты (в них следует использовать макрос `assert`), после чего функция инициализации сообщает об ошибке при загрузке модуля.

Для более удобной отладки кода ядра, не связанного с написанием драйверов физических устройств, можно использовать так называемый User Mode Linux (UML)[9], который представляет собой вариант ядра, работающий как процесс пользователя поверх другого ядра. Дистрибутив Debian содержит пакеты для простой установки UML.

Для этих целей можно применить виртуальные машины, например, QEmu или Virtualbox, легко устанавливаемые в Debian GNU/Linux. С помощью QEmu можно проверить работоспособность созданного кода на архитектурах, отличных от архитектуры используемого компьютера.

6. ПРИМЕР МОНИТОРИНГА СИСТЕМНОГО ВЫЗОВА

6.1. Постановка задачи

Рассмотрим простейший пример программирования для ядра Linux. Целью примера является создание механизма информирования пользователя о числе системых вызовов kill. Для этого необходимо решить следующие задачи:

1) добавить в исходные тексты ядра механизм уведомления о системном вызове, собрать новое ядро и установить его;

2) создать модуль ядра, использущий этот механизм и реализующий основную логику работы системы, и собрать модуль с помощью системы сборки нового ядра;

[9]Веб-узел проекта: `http://user-mode-Linux.sourceforge.net`.

3) создать пользовательское приложение с графическим интерфейсом для демонстрации работы модуля.

Для создания интерфейса пользователя можно, например, использовать язык Python, библиотеку PyGTK и библиотеку построения графиков matplotlib. Поскольку эта задача не относится к теме пособия, то ее решение не приводится. Следует отметить, что такое приложение должно читать данные из файла `/proc/kill_hook`. С целью демонстрации возможно также создание графического приложения, вызывающего стандартные консольные программы для загрузки и выгрузки созданного модуля ядра.

6.2. Перехват информации о событиях ядра

Для решения задачи 1 (см. разд. 6.1) необходимо выбрать в исходных текстах ядра функцию, в которую следует добавить механизм уведомления. Первый кандидат на эту роль – функция — обработчик системного вызова `sys_kill`. В других случаях может появиться необходимость устанавливать перехват событий более «глубоко», поскольку отдельный системный вызов — это часто лишь один из возможных инициаторов события.

Для перехвата системного вызова следует внести изменения в файлы `signal.c` и `signal.h`:

```
/* include/linux/signal.h */
typedef void (*kill_hook_t)(struct siginfo *info,
    int pid, int sig);
kill_hook_t set_kill_hook(kill_hook_t hook);
/* kernel/signal.c */
static kill_hook_t kill_hook;
kill_hook_t set_kill_hook(kill_hook_t hook)
{
    kill_hook_t old_hook = kill_hook;
    kill_hook = hook;
    printk(KERN_INFO "kill hook: %p", hook);
    return old_hook;
}
EXPORT_SYMBOL_GPL(set_kill_hook);
asmlinkage long sys_kill(int pid, int sig)
```

```
{
    struct siginfo info;
    info.si_signo = sig;
    info.si_errno = 0;
    info.si_code = SI_USER;
    info.si_pid = task_tgid_vnr(current);
    info.si_uid = current->uid;
   /* Опасность гонок при снятии обработчика! */
    if (kill_hook)
        kill_hook(&info, pid, sig);
    return kill_something_info(sig, &info, pid);
}
```

В функции `sys_kill` существует опасность «гонок», если модуль с обработчиком будет выгружен в момент обработки сигнала. Для решения этой проблемы следует использовать семафор, синхронизирующий функции `sys_kill` и `set_kill_hook`.

Функция `set_kill_hook` отмечена в языке как экспортируемый символ с помощью макроса `EXPORT_SYMBOL_GPL`, что позволяет вызывать ее только из модулей, лицензированных по GPL.

После внесения изменений следует получить патч и проверить его на соответствие стилю кодирования ядра. В случае корректного кода проверка выдаст единственную ошибку об отсутствии подписи разработчика, так как правила разработки ядра требуют, чтобы каждое изменение ядра имело своего автора.

6.3. Создание модуля с основной логикой работы

Модуль регистрирует обработчик инфомации о вызове и создает файл `/proc/kill_hook`, через который модуль сообщает число зарегистрированных системых вызовов с момента последнего чтения этого файла. Для хранения счетчика используется атомарная переменная.

Ниже приведен полный исходный код этого модуля:

```
#include <linux/module.h>
#include <linux/signal.h>
#include <linux/proc_fs.h>
MODULE_LICENSE("GPL");
struct proc_dir_entry *proc_file;
```

```
const char *proc_entry = "kill_hook";
/* Атомарная переменная */
static atomic_long_t count;
/* Функция передает через procfs текущее
 * значение счетчика и сбрасывает его. */
static ssize_t procfile_read(char *buffer,
    char **buffer_location, off_t offset,
     int buffer_length, int *eof, void *data)
{
    int len = 0;
    long value;
    value = atomic_long_read(&count);
    /* Опасность гонок: нельзя просто сбросить
     * счетчик в ноль без мьютекса. */
    atomic_long_add(-value, &count);
    *eof = 0;
    /* Если не первый запорос -- сообщаем
     * о конце ƒайла */
    if (offset > 0) {
            *eof = 1;
        return 0;
    }
    /* Записываем данные в буƒер */
   len = sprintf(buffer, "count = %ld\n", value);
    return len;
}
/* Счетчик системного вызова */
void counter(struct siginfo *info,
    int pid, int sig)
{
    atomic_long_inc(&count);
}
/* Инициализация модуля */
int hook_init(void)
{
    int rv = 0;
    atomic_long_set(&count, 0);
    proc_file = create_proc_entry(proc_entry,
```

```
        0644, NULL);
    if (proc_file == NULL) {
        rv = -ENOMEM;
        goto error;
    }
    proc_file->read_proc = procfile_read;
    proc_file->write_proc = NULL;
    proc_file->owner = THIS_MODULE;
    proc_file->mode = S_IFREG | S_IRUGO;
    proc_file->uid = 0; proc_file->gid = 0;
    printk(KERN_INFO "%s was created\n",
           proc_entry);
    set_kill_hook(counter);
error:
    return rv;
}
/* Завершение работы модуля */
void hook_exit(void)
{
    set_kill_hook(NULL);
    remove_proc_entry(proc_entry, proc_file);
   printk(KERN_INFO "%s removed\n", proc_entry);
}
module_init(hook_init);
module_exit(hook_exit);
```

Для получения информации от модуля можно воспользоваться, например, следующей шелл-программой:

```
$ while [ 1 ]; do \
      cat /proc/kill_hook; sleep 5; done
```

ЛИТЕРАТУРА

1. *Керниган Б.* Язык программирования Си : пер с англ. / Б. Керниган, Д. Ритчи. М. : Вильямс, 2006. 304 с.

2. *Таненбаум Э.С.* Операционные системы. Разработка и реализация / Э.С. Таненбаум, А. Вудхалл. СПб. : Питер, 2007. 704 с.

3. *Робачевский А.М.* Операционная система Unix / А.М. Робачевский. СПб.: БХВ-Петербург, 2007. 656 с.

4. *Corbet J.* Linux Device Drivers / J. Corbet, A. Rubini, G. Kroah-Hartman. Sebastopol : O'Reilly, 2005. 636 p. `http://lwn.net/Kernel/LDD3`

5. *Kroah-Hartman G.* Linux Kernel in a Nutshell / G. Kroah-Hartman. Sebastopol : O'Reilly, 2006. 198 p. `http://lwn.net/kroah/lkn`

6. *Burian M.* The Linux Kernel Module Programming Guide / M. Burian, P.J. Salzman. `http://tldp.org/LDP/lkmpg/`

7. *Бовет Д.* Ядро Linux : пер. с англ. / Д. Бовет, М. Чезати. М. : BHV, 2007. 1104 с.

8. *Лав Р.* Разработка ядра Linux : пер. с англ. / Р. Лав. М. : Вильямс, 2006. 448 с.

ОГЛАВЛЕНИЕ

Учебное издание

Крищенко Всеволод Александрович
Рязанова Наталья Юрьевна

ОСНОВЫ ПРОГРАММИРОВАНИЯ В ЯДРЕ ОПЕРАЦИОННОЙ СИСТЕМЫ GNU/LINUX

Редактор *О.М. Королева*
Корректор *М.А. Василевская*
Компьютерная верстка *В.И. Товстоног*

Подписано в печать 10.06.2010. Формат 60×84/16.
Усл. печ. л. 2,09. Тираж 100 экз. Изд. № 149.
Заказ

Издательство МГТУ им. Н.Э. Баумана.
Типография МГТУ им. Н.Э. Баумана.
105005, Москва, 2-я Бауманская ул., 5.