Основы программирования в ядре операционной системы GNU/Linux

Учебное пособие. Черновик

Крищенко В. А., Рязанова Н. Ю.

29 сентября 2008 г.

Аннотация

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 пересборка 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 третьего курса кафедры «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н. Э. Баумана.

Необходимость в пособии обусловлена трудностями, с которыми студент вникает в LDD, LKD и другие оригинальные источники (замечание Крищенко В. А.).

Содержание

Введение

Создание программного кода, работающего в ядре операционной системы, является достаточно своеобразной задачей. Причиной этому является как непривычный для студента характер работы, заключающейся в интеграции его кода в уже существующую программу ядра, так и невозможность использовать привычные средства разработки, отладчики и программные библиотеки, предназначенные для прикладных программ.

В результате первый месяц или даже два студент пытается понять, что к чему, а времени на курсовой проект выделено немного (замечание Крищенко В. А.).

Данное учебное пособие содержит начальные сведения, необходимые для разработки в ядре операционной системы GNU/Linux (в дальнейшем ядро Linux называется просто «ядро»). Предполагается наличие у читателя знания языка программирования Си [1], общего представления об интерпретаторе командной строки и основных служебных программах GNU [1], компиляторе GNU C, системе сборки GNU Make, а так же основными понятиям из теории операционных систем (например, по [2]).

Материал пособия ориентируется на операционную систему GNU/Linux в вариантах Debian или Ubuntu, но в целом верен для любых операционных систем, использующих ядро Linux и средства разработки GNU. Примеры проверены для версии ядра Linux 2.6.26 и версии дистрибутива Debian GNU/Linux 5.0. *Пока пособие напечают – 5.0 точно выйдет (замечание Крищенко В. А.).* Предполагается использование GNU Bourne-Again Shell в качестве командного интерпретатора. *Использую башизм $(command), чтобы не мучаться с «‘», тем более что $(...) –нагляднее (замечание Крищенко В. А.).*

В качестве литературы для дальнейшего изучения программирования в ядре можно посоветовать прежде всего книги, которые легально и бесплатно доступны в интернете: [3] [2], [4] [3], а также не выходившую в печатном виде работу [5] [4]. Следует упоминуть также посвященные ядру Linux книги [6, 7], выходившие на русском языке.

Несмотря на разнообразие посвященной ядру Linux литературы, единственным заведомо актуальным источником информации о ядре являются исходные тексты используемой версии ядра. Одной из причин этому является постоянное изменение внутренних структур данных ядра и заголовков функций даже при смене номера минорной версий ядра. В силу этого исходный код, работоспособный в версии 2.6.9, в версии 2.6.25 обычно не проходит даже этап компиляции. Данная проблема значительно усложняет разработку сторонних драйверов. *С другой стороны, она же приводит к определенным сложностям при сдаче прошлогодних курсовых проектов, что выглядит скорее как достоинство при использовании ядра Linux в учебных целях (замечание Крищенко В. А.).*

Для ядра Linux практически не существует какой-либо программной документации, отличной от самих исходных текстов ядра с комментариями и содержимого каталога `Documentation` в архиве исходных текстов ядра. Таким образом, студенту для успешной работы необходимо научиться работать с чужим исходным кодом. Для удобного поиска как мест определения, так и мест использования различных глобальных символов (функций, макросов, типов данных и глобальных переменных) может использоваться специализированная поисковая система по исходным текстам ядра Linux Cross Reference [5].

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

[2] Доступна по адресу: `http://lwn.net/Kernel/LDD3/`.

[3] Доступна по адресу: `http://www.kroah.com/lkn/`.

[4] Версия для ядра 2.6 доступна по адресу `http://tldp.org/LDP/lkmpg/2.6/`.

[5] Находящаяся по адресу `http://lxr.linux.no/linux/`.

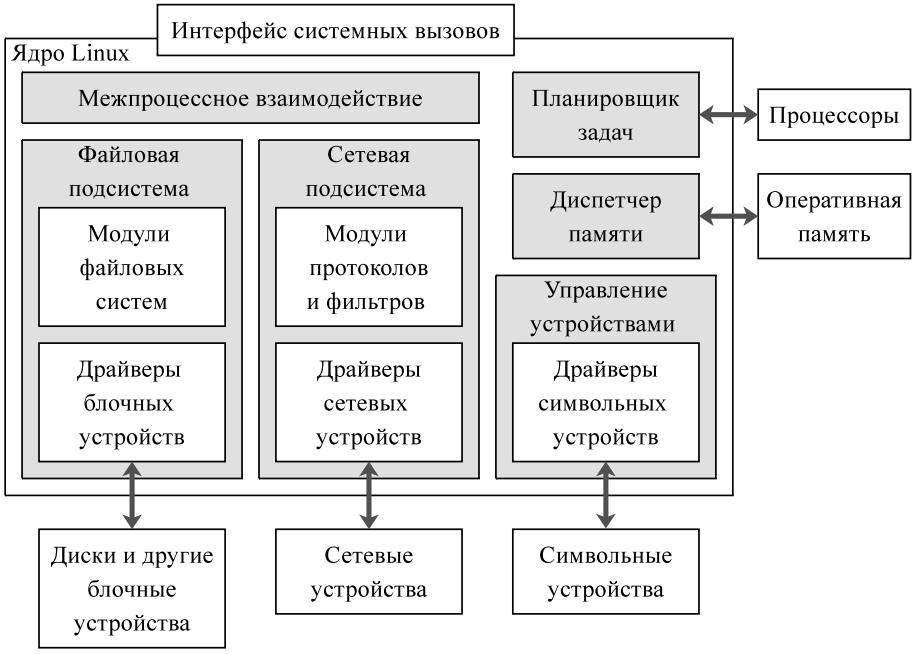

Рис. 1. Основные компоненты ядра Linux

Текущее ядро имеет лицензию GNU GPL 2.0. Linux является зарегистрированной торговой маркой Линуса Торвальдса.

В пособии оригинальные английские термины даны курсивом, например: *spinlock*. Имена команд, файлов и каталогов, функций, а так же фрагменты исходного кода выделяются моноширным шрифтом. Команды, вводимые пользователям, начинаются с символа-приглашения:

```
$ make install
```

1. Начало работы

1.1. Краткие сведения о ядре

Linux является ядром многопользовательской многозадачной операционной системы и поддерживает (с ограничениями) стандарты POSIX и Single UNIX Specification [6], а так же ряд собственных программных интерфейсов.

Ядро состоит из следующих основных составляющих: планировщик процессов, менеджер памяти, подсистема ввода-ввода и драйверы устройств, сетевая подсистема, виртуальная файловая система, подсистема межпроцессного взаимодействия и других (рис. 1). 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,

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

драйверы устройств или код, ответственный за конкретные файловые системы. Модуль оформляется в виде отдельного файла. На этапе загрузки ядра необходимые модули могут загружаться с диска или из временного диска в памяти, организуемого при помощи начального загрузчика ОС.

1.2. Подготовка рабочего места

В дальнейшем будет предполагаться, что работа с исходным кодом ядра ведётся в каталоге, имя которого хранится в переменной окружения `$SYSPROG`, а рабочая версия файлов находится в каталоге `$SRC`. Экспортируем переменные окружения и создадим рабочий каталог для работы с исходными текстами ядра.

```
$ export SYSPROG=~/sysprog
$ export SRC=${SYSPROG}/linux-source
$ mkdir -p ${SYSPROG}
```

Для работы с исходными текстами ядра и для пересборки ядра желательно иметь около трёх гигабайт свободного дискового пространства. Для установки компилятора Си и системы сборки в Debian GNU/Linux можно использовать следующую команду.

```
$ sudo apt-get install build-essential
```

Если предполагается создавать только модули для текущего ядра дистрибутива, то достаточно установить заголовочные файлы ядра.

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

Если предполагается изучение исходных текстов ядра или его модифиация, то следует поставить всё дерево его исходных текстов. Ядро Linux разрабатывается с использованием системы управления версиями Git, но для описанных в пособии задачах исходные тексты проще всего взять архив исходников из используемого дистрибутива. Для установки исходных текстов ядра дистрибутива следует установить пакет `linux-source`, и затем развернуть архив из `/usr/src/linux` в желаемый каталог.

```
$ export VER=$(uname -r | cut -f1 -d-)
$ sudo apt-get install linux-source-$VER
$ cd $SYSPROG
$ # Создание двух копий исходных текстов ядра
$ tar xf /usr/src/linux-source-$VER.tar.bz2
$ mv linux-source-$VER linux-source
$ tar xf /usr/src/linux-source-$VER.tar.bz2
$ mv linux-source-$VER linux-source.orig
```

После выполнения этих команд в каталоге `$SYSPROG` появятся два подкаталога:

- каталог `linux-source` содержит изменяемую версию исходных текстов, в дальнейшем на него мы будем ссылаться как на `$SRC`;
- каталог `linux-source.orig` содержит исходную версию ядра, которая необходима для создания патча.

Если необходимо внести изменения в какую-то конкретную версию ядра, то ее исходные тексты следует получить с сервера `http://kernel.org/`. При написании данного пообия использовалась версия 2.6.26.5, следующие команды так же создают два дерева.

```
$ export VER=linux-2.6.26.5
$ cd $SYSPROG
$ wget eu.kernel.org/pub/linux/kernel/v2.6/${VER}.tar.bz2
$ tar xf ${VER}.tar.bz2 && mv ${VER} linux-source
$ tar xf ${VER}.tar.bz2 && mv ${VER} linux-source.orig
```

Для написания исходного кода режима ядра можно использовать как текстовые редакторы с подсветкой синтаксиса и интергацией 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 производится 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 ресурсоёмкая 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 определением функций, макросов и типов данных через контекстное меню редактора среды или при помощи поля поиска символов.

1.3. Обзор исходных текстов ядра

Ядро Linux поддерживает в настоящее время более двух десятков архитектур, как 32-х, так и 64-х 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 исключлючительно на языке программирования Си.

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

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

Следует отметить, что полная аппаратная архитектура задается многочисленными параметрами в конфигурации. Например, за архитектуру x86 отвечают три основных параметра: `CONFIG_X86`, `CONFIG_X86_32` (указывает на 32-х разрядную архитектуру) и

CONFIG_X86_64 (указывает на 64-х разрядную архитектуру), и множество дополнительных, например, CONFIG_X86_SMP отвечает за поддержку многопроцессорной архитектуры. Таким образом, аппаратные архитектуры i386 и amd64 в современных версиях ядра объединены в одну – x86 [7]. При необходимости использовать различающийся код для каждой из них используются директивы условной компиляции, как показано ниже.

```
# ifdef CONFIG_X86_32
#  include "unistd_32.h"
# endif
# ifdef CONFIG_X86_64
#  include "unistd_64.h"
# endif
```

Исходные тексты ядра разделены по своему назначению на несколько каталогов, из которых представляют наибольший интерес следующие:

- `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 текстов. Данная операция может понадобиться в следующих случаях:

- при внесение изменений в исходный текст ядра;
- при изменении конфигурации ядра.

Обычно часть модулей ядра включается в загрузочный временный 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 текстов управляется конфигурацией ядра, которая хранится в файле `$SRC/.config`. В качестве начальной конфигурации можно взять, например, текущую конфигурации ядра.

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

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

Для идентификации создаваемой версии ядра следует изменить его версию, заданную в файле `$SRC/Makefile`. Рекомендуется поменять поле EXTAVERSION. Следующая команда меняет это поле на «-test» (конечно же, можно использовать и текстовый редактор).

```
$ sed "s/^EXTR.*$/EXTRAVERSION=-test/" -i $SRC/Makefile
```

Далее будем считать, что в переменной VERSION задается полная версия нового ядра.

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

В файле сборки ядра заданы несколько целей, позволющих производить типичные операции с ядром при помощи команды `make`, отданной в каталоге `$SRC`. Перечислим основные из них.

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

2) Для изменения конфигурации ядра обычно используется комманда `make menuconfig`. При помощи этой команды можно создать компактное и быстро собирающееся ядро, но делать это следует с большой осторожнотстью.

3) После изменения конфигурации или исходного кода ядра его следует пересобрать командой `make`.

4) Для установки ядра в каталог `boot`, а его штатных модулей и системы сборки модулей в каталог `libmodules` применяется команда `make install`.

5) Команда `make modules_install` устанавливает модули, а также систему сборки новых модулей без установки самого ядра.

Для загрузки ядра требуется, помимо установки ядра, создать образ временного загрузочного диска initrd, который считывается загрузчиком ОС, временно монтируется в качестве корневой файловой системы и затем используется для подгрузки 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 текущем ядре дистрибутива можно просто взять его образ, но в общем случае для создания образа диска используется специмальная программа, например `mkinitramfs` из пакета *initramfs-tools*.

```
$ cd ${SRC}
$ # Создание initrd.img
$ mkinitramfs -o initrd.img-$VERSION $VERSION
$ # Копирование образа в каталог начальной загрузки
$ sudo cp initrd.img-$VERSION /boot
```

После установки ядра и образа загрузочногго диска следует создать в файлt загрузчика раздел для загрузки нового ядра с использованием созданного образа [8]. 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. В случае Debian изменения в `/boot/grub/menu.lst` лучше производить вне автоматически поддерживаемого системой списка вариантов загрузки.

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

2. Основы программирования для ядра Линукс

2.1. Оформление кода

Исходные тексты ядра операционной системы GNU/Linux состоят главным образом из кода на языке Си стандарта C99, причем в ряде случаев используются специфичные возможности компилятора 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 [9].

Для проверки соблюдения принятого в ядре стандарта кодирования существует входящая в архив исходных текстов ядра программа `checkpatch.pl`, которая может проверять как патчи, так и отдельные файлы (ключ `file`). Она находится в каталоге `$SRC/scripts`.

Документы, описывающие стандарты кодирования, прилагаются к исходным текстам 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 текстах ядра используется только табуляция с предположением, что она представлена в редакторах кода восемью пробелами[10].

Поскольку в языке Си нет исключений, то в коде ядра для индикации ошибки используется результат функции:

- 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 отсутвие ошибки, ненулевое – некоторый код ошибки.

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

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

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

[10] В данном пособии табуляция представлена четырьмя пробелами из-за меньшей, по сравнению с экраном, ширины печатного листа. Кроме того, возможные иные отхождения от стандарта.

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

Программы, работающие в режиме 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 текстов и новая загрузка системы, в то время как для создания модуля ядра достаточно наличие заголовочных файлов ядра и системы сборки модулей, а загрузка и выгрузка модулей может производиться «на лету». Разработка модулей гораздо более удобна, но далеко не все желаемые изменения можно выполнить из модуля. Например, установить перехватчик каких-либо событий ядра невозможно без внесения некоторых изменений в его исходный код. Таким образом, учебная работа может включать в себя отдельный модуль ядра и некоторые изменения в исходниках ядра, называемые «патчем» (*patch*).

Стандартной формой для представления изменений в исходные тексты ядра является патч, который представляет разницу между исходным и изменённым набором исходных текстов ядра. Патчи создаются с помощью программы `diff` в соответствии с документацией в файле `Documentation/patches`, а применяются к исходным текстам («накладываются») с помощью программы `patch`.

Файл `Documentation/dontdiff` содержит шаблоны имен файлов, которые не подлежат сравнению. С его помощью можно создать патч сразу для всех изменённых файлов следующими командами.

```
$ 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 в текстовом редакторе и удалить информацию о «лишних» файлах, которая там может находится после указанной процедуры. Для проверки соотвествия патча стандартам кодирования, принятым в ядре Linux, используется программа `scripts/checkpatch.pl`.

Для применения патча следует зайти в каталог с исходными текстами, которые необходимо изменить («пропатчить»), и выполнить примерно следующую команду:

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

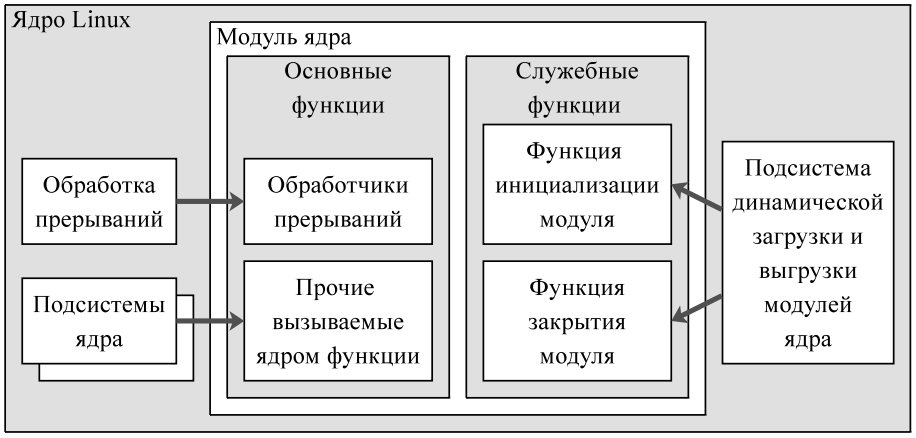

Рис. 2. Взаимодействие модуля и ядра

2.3. Создание модулей ядра

Модуль ядра представляет собой объектный файл, загружаемый в адресное пространство ядра и динамически линкуемый с ядром (рис. 2). Код модуля выполняется с теми же привилегиями, что и остальной код ядра. Основное ограничение модулей заключается в том, что они не могут использовать функции ядра, которые не экспортируются.

Модуль может подгружаться в момент загрузки ядра из временного диска начальной загрузки или из каталога `/lib/modules/$(uname -r)` после монтирования корневого каталога. Для операций с модулями существует несколько специальных служебных программ.

1) Для добавления и удаления постоянно загружаемых модулей используется программа `modprobe`.

2) Модуль может подгружаться и выгружаться в ходе работы системы программами `insmod` и `rmmod`. Произведенные ими изменения не сохраняются после перезагрузки ядра.

3) Служебная программа `lsmod` отображает список загруженных модулей (используйте `grep` для фильтрации).

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

Создание объектного файла модуля значительно отличается от создания объектных файлов пользовательских программ, поэтому для упрощения это задачи используется система сборки модулей ядра. Для её использования следует вызвать её файл сборки из файла сборки модуля. Система сборки модулей организует двухэтапный процесс, упрощённо показанный на рис. 3.

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 при помощи системы сборки именно

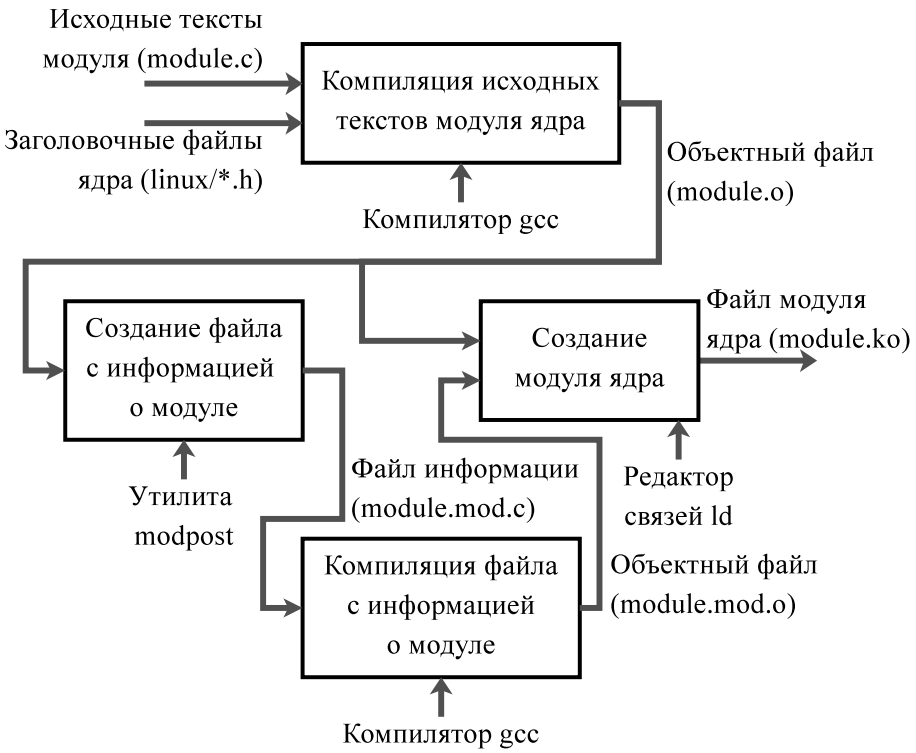

Рис. 3. Сборка модуля ядра

того же ядра, с которым он затем будет работать, иначе попытка загрузить модель приведет к ошибке (при загрузке модуля проверяются полные версии ядер).

Исходный текст 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. Как видно из комментариев, он сначала вызывает систему сборки модуля в цели `default`. Затем этот же файл будет включён системой сборки модклей (переменная `$(KERNELRELEASE)` при этом уже будет установлена), и дожен сообщить ей имя модуля. В данном простейшем случае модулю будет соотвествовать единственный файл исходного кода с тем же именем (`test_module.c`).

```
# 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
```

Две функции модуля ядра являются выделенными: функция инициализации и выхода. После загрузки модуля и его связывания с остальным кодом ядра, 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 способо 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 полученым объектным файлам.

3. Программные интерфейсы ядра

3.1. Служебные функции ядра

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

1) Функции для обработки строк, аналогичные стандартной библиотеке языка Си. Их прототипы доступны через заголовочный файл `linux/string.h`.

2) Функции для обработки областей памяти. Их прототипы доступны через `linux/string.h`.

3) Функции sprintf, sscanf и им подобные, а так же функции отладочной печати доступны через `linux/kernel.h`.

4) Функции динамического выделения памяти, прототипы которых находятся в `linux/slab.h`.

5) Функции и макросы для создания списков доступны через `linux/lists.h` и `linux/plists.h` (списки с приоритетами).

6) Функции организации синхронизации и работы с нитями (*threads*) доступны после включения файла `linux/sched.h`.

7) Функции и макросы поддержки модулей доступны в файле <linux/module.h>.

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

Ряд вспомогательных функций соответствую стандартным названием и назначением (например, strlen) или имеют букву «k» (от *kernel*), напрмер kmalloc, но многие сервисные функции не имеют аналогов в стандартной библиотеке (например, memparse).

Отдельно стоят функции логирования (printk и другие из linux/kernel.h), которые передают информацию службе klogd и являются важным инструментом отладки внутриядерного кода. Использование функции printk выглядит примерно следующим образом.

```
printk(KERN_INFO "Blink: period = %d\n", blink_period);
```

Используемый 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 (выделения памяти и ее заполнения нулями), для освобождения – 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 так же использует данный механизм кеша.

Кроме требуемого объема памяти, все функции выделения памяти в ядре принимают так же параметр, определяющий требуемое поведение ядра при выделении памяти и свойства выделенного блока. Существует множество флагов, определяющих 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 пройзойдет смены 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 используется при выделении значительных объемов памяти в некритических случаях. При его использовании вызывающая нить может быть заблокирована, если это необходимо для выделения памяти. Например, дисковый кеш может быть сброшен на диск для высвобождения памяти или данные какого-либо процесса помещенны в область подкачки. Если же и подобные операции не приводят к освобождению требуемого количества памяти, функция вернет нулевой указатель.

Ядро версии 2.6 предлагает механизмы для организации связанных списков (list.h) и связанных списков с приоритетами (plist.h), причем создание аналогичных механизмов с аналогичной функциональностью программистом крайне нежелательно. В списка ядра вместо часто используемой в учебниках (и весьма неэффективной) структуры, содержащей указатель на данные и на соседние элементы списка, используется непосредственное

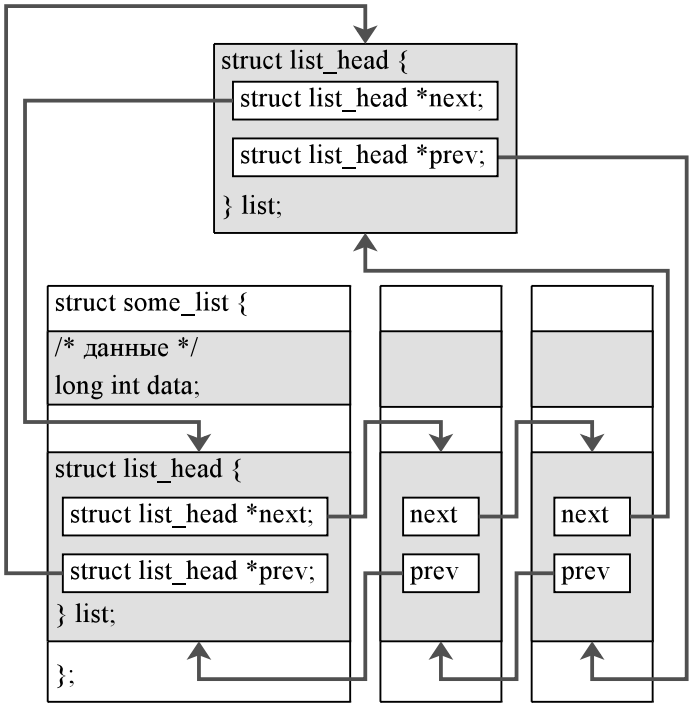

Рис. 4. Связанные списки ядра

добавление структуры типа `list_head` в качестве поля в структуру, содержащую данные (рис. 4).

Используемая в ядре реализация списков позволяет выделять на каждый элемент списка только один блок памяти, что позволяет значительно сократить накладные расходы на организацию списков, и при этом дать набор функций и макросов для манипулирования с любыми списками.

Для хранения «головы» и «хвоста» списка используется отдельная структура, для её 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 используется фунция `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 *iter, *iter_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(iter, &list) {
        item = list_entry(iter, struct data, list);
        printk(KERN_INFO "[LIST] %d\n", item->n);
    }
out:
    list_for_each_safe(iter, iter_safe, &list) {
        item = list_entry(iter, struct data, list);
        list_del(iter);
        kfree(item);
        }
}
```

Для обхода списка с целью удаления используется макрос `list_for_each_safe`, который использует дополнительную переменную для того, чтобы можно было безопасно удалять элементы списка в ходе его обхода.

Макрос `list_entry` позволяет получить адрес структуры по адресу ее поля. Он эквивалентен макросу `container_of`, описанному в `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 `include/linux/stddef.h` и позволяет получить 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

Системных вызовы (*syscalls*) являются программным интерфейсом ядра, предназначенным исключительно для использования процессами в пространстве пользователя через архитектурно-зависимый интерфейс, обычно использующий прерывания.

Для добавления нового системного вызова необходимо пересобрать ядро, предварительно добавив информацию о новом вызове в таблицы системных вызовов. Расположение и вид таблицы системных вызовов является зависимым от архитектуры системы. В случае архитектуры x86 в версии ядра 2.6.26 она располагется в файле *syscall_table_32.S*.

Поскольку механизм системных вызовов зависит от аппаратной архитектуры, то для упрощения использования нового системного вызова существует системный вызов `sys_syscall` и вызвающая его функция `syscall` в стандартной библиотеке. Эта функция позволяет сделать системный вызов по его номеру.

Таблица системных вызовов в текущих версиях ядра Линукс не экспортируется и, следовательно, недоступна для модулей ядра. Для решения это проблемы «силой» к ядру иногда применялся патч, экспортирующий её и позволяющий таким образом установить указатель на свою функцию в таблицу системных вызовов. Однако, делать так не следует ни в коем случае по ряду причин, таблица системных вызовов не должна меняться после сборки ядра. Для решения задачи перехвата вызовов, если она стоит, лучше создать отдельный механизм уведомления, как будет показано далее.

3.4. Обмен данными с прикладным программам

Для обмена данными между ядром и пользовательскими программами используются системные вызовы. Однако, если прикладным программам нужно получать из ядра какие-либо данные, то добавление для этой задачи отдельных системных вызовов не удобно. Если принять во внимание парадигму «в Unix (почти) всё – файл», то становится ясно, что прикладным программам должна быть предоставлена возможность использовать системные вызовы работы с файлами для реализации любого информационного обмена с ядром. Для этого в GNU/Linux может использоваться файловая система ProcFS, связанная с каталогом `/proc`. Данная файловая система позволяет установить функции (*callbacks*), вызываемые при чтении или записи 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 разделенные проблеами или табуляциями. Такой файл легко обрабатывать при помощи стандартных служебных программ `grep`, `cut`, `wc`, `awk` и других.

Для работы с ProcFS доступны два интерфейса: основной, описанный в файле `proc_fs.h`, и более новый и более удобный в ряде случаев интерфейс последовательностей (*sequence*), описанный в файле `seq_file.h`. Достаточно подробно эти интерфейсы рассмотрены в [5].

Альтернативой применению ProcFS может является 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 [3].

4. Нити и синхронизация в ядре

4.1. Нити внутри ядра

Ядро версий 2.6 поддерживает симметричную многопроцессорную архитектуру (SMP) и внутриядерные нити (*kernel threads*). Для создания нити используется функция `kernel_thread`, доступная после директивы `#include <linux/sched.h>` и описанная в файлах `processor.h`.

В силу этого даже при наличии одного процессора выполнение нити может быть прервано, и управление может быть передано другой нити. Поэтому для ликвидации про-

блемы гонок (*race conditions*) доступ к любым программным или аппаратным ресурсам, к которым теоретически может осуществляться доступ более, чем одной нитью, должен быть упорядочен (синхронизирован) явным образом. В частности, это касается изменения любых глобальных данных.

Для решения задачи синхронизации в ядре Linux существует множество механизмов синхронизации, обычно доступные после включения файла `linux/sched.h`, такие как:

- переменные, локальные для каждого процессора (*per-CPU variables*), интерфейс которых 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 завершения ее выполнения. В ядре нет аналога функции ожидания завершения нити, вместо нее требуется явно использовать механизмы синхронизации.

Использование для задачи ожидания какого-либо события обычного семафора не рекомендуется: в частности, реализация семафора оптимизирована исходя из предположения, что обычно они открыты. Поскольку в данном случае не идет речь о критических секциях, то для этой задачи 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 её в состояние готовности выполнять работу.

Следующий пример демонстрирует запуск нити и ожидание завершения его выполнения.

```
static int thread(void * data)
{
    struct completion *finished = (struct completion *)data;
    complete(finished);
    return 0;
}
int test_thread(void)
{
    pid_t pid;
    DECLARE_COMPLETION(finished);
    /* флаги описаны в sched.h */
    pid = kernel_thread(thread, &finished, CLONE_FS);
```

```
    wait_for_completion(&finished);
    return 0;
}
```

4.3. Семафоры, мьютексы и спин-блокировки

Семафор в ядре представляет собой механизм, позволяющий ограничить количество нитей, одновременно выполняющихся в некоторой области кода. Это количество называется значением семафора. Мьютекс (от *mutual exclusion*) представляет собой семафор со значением, равным единице. Для обозначения открытия и закрытия семафоров в ядре используются термины *down* и *up* соответственно.

Семафоры в ядре подобны POSIX-семафорам, используемых прикладными программами, но имеют несколько другой интерфейс. Существует очевидное ограничение на использование семафоров в ядре: их невозможно использовать в том коде, который не должен «уснуть», например при начальной обработке прерываний. Для синхронизации в последнем случае используется спин-блокировка (*spin lock*), использующая простое ожидание в цикле. Кроме того, неудачная попытка входа в критическую секцию при использовании 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 использовать спин-блокировку, основанную на простом ожидании в цикле.

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

Особым, но часто встречающимся, случаем синхронизации являются случай так называемых «читателей» и «писателей». Читатели только читают состояние некоторого ресурса, и поэтому могу осуществлять к нему параллельный доступ. Писатели изменяют состояние ресурса, и в силу этого писатель дожен иметь к ресурсу монопольный доступ, причем чтение ресурса в этот момент времени так же должно быть заблокированно.

Для реализации писателей и читателей в ядре Linux существуют специальные версии семафоров и спин-блокировок. Мьютексы реального времени не имеют реализации для случая читателей и писателей, поэтому разработчик должен сделать осознанный вьбор механизма синхронизации.

```
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 *iter;
    down_read(&rw_sem);
    list_for_each(iter, &list) {
        item = list_entry(iter, 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 задачи ресурс представляет собой единственную целочисленную глобальную переменную, то использование семафоров представляется избыточным. Однако, отсутствие блокировок приводит к тому, что даже результат выражения `a++` с глобальной переменной на некоторых архитектурах может приводить к неожиданному результату в случае параллельно выполняющихся задач.

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`. Использование атомарных переменных рассмотрено в конце пособия в главе с примером.

5. Отладка внутриядерного кода

Одной из очевидной трудностей программирования в режиме ядра является невозможность использования отладчика прикладных программ. Для решения этой проблемы мож-

но использовать специальный отладчик ядра, для чего в частности требуется пересобрать ядро, и использовать отладку в виртуальной машине. Однако, использование отладчика не решает всех проблем создания надежных программ, поэтому далее будет рассмотрены вопросы отладочной печати, проверки условий и модульного тестирования.

Одним из основных методов отладки программ является отладочная печать (ведение журнала работы) и проверка условий (конструкция *assert*). Для отладочной печати используют функцию `printk`. Для удаления малозначимых сообщений из рабочей версии можно использовать директивы препроцессора.

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

Сообщения не должны использовать символы вне 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 `/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 вопросом, как следует оформлять тесты, ведь создаваемый код, вообще говоря, не может быть скомпилирован для работы в пользовательском режиме. Для модульного тестирования части кода можно использовать модуль примерно со следующий функций 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\n", 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. Следует организовать автоматическую генерацию когда этого модуля, ища в объектных файлах функции с именами `test_*`.

Для более удобной отладки кода ядра, не связанного с написанием драйверов физических устройств, можно использовать так называемый User Mode Linux (UML) [11], который представляет собой вариант ядра, работающий как процесс пользователя поверх другого ядра. Дистрибутив Debian содержит пакеты для простой установки UML. Так же для этих целей можно использовать виртуальную машину, например QEmu или Virtualbox, легко устанавливаемые в Debian GNU/Linux. С помощью QEmu можно проверить работоспособность созданного кода на крхитектурах, отличных от архитектуры используемого компьютера.

6. Пример: мониторинг системного вызова

6.1. Постановка задачи

В этой главе будет рассмотрен простейший пример программирования для ядра ОС. Целью примера является создания механизма информирования пользователя о числе системых вызовов kill. Для этого необходимо сделать следующее.

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.

3) Создать пользовательское приложение с графическим интерфейсом для демонстрации работы модуля. *Демонстрация работы модуля при помощи команд типа `tail -F /proc/test` может не произвести должного впечатления (замечание Крищенко В. А.).*

Последнюю задачу можно легко решить, используя, например, язык Python, библиотеку PyGTK для создания интерфейса пользователя и библиотеку построения графиков matplotlib. Поскольку эта задача не относится к теме пособия, то ее решение здесь приведено не будет, следует лишь отметить, что такая программа должна читать данные из `/proc/kill_hook`, а для тестирования модуля вполне достаточно читать из этого файла данные при помощи программ типа `grep` или `tail`. С целью демонстрации возможно также создание графического приложения, вызывающего стандарные консольные программы для загрузки и выгрузки созданного модуля ядра.

6.2. Перехват информации о событиях ядра

Для решения первой задачи необходимо выбрать в исходных текстах ядра фунцию, в которую следует добавить механизм уведомления. Первый кандидат на эту роль – функция-обработчик системного вызова `sys_kill`, вызывающаяся непосредсредственно

[11] `http://user-mode-linux.sourceforge.net/`

из обработчика системного вызова, которая и будеит изменена. При другой решаемой задаче обычно необходимости устанавливать перехват событий более «глубоко», поскольку отдельный системный вызов является часто лишь одним из возможных инициаторов представляющего интерес события.

Для перехвата внесём изменения в файлы `signal.c` и `signal.h` [12]

```
/* signal.h */
typedef void (*kill_hook_t)(struct siginfo *info,
    int pid, int sig);
void set_kill_hook(kill_hook_t hook);
/* signal.c */
static kill_hook_t kill_hook;
void 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;
    if (kill_hook)
        kill_hook(&info, pid, sig);
    return kill_something_info(sig, &info, pid);
}
```

Функция `set_kill_hook` помечена как символ, экспортируемый ядром, при помощи макроса `EXPORT_SYMBOL_GPL`, что позволит вызывать ее только из модулей, лицензированных по лицензии GPL.

6.3. Создание модуля с основной логикой работы

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

```
#include <linux/module.h>
#include <linux/signal.h>
```

[12] К сожалению, привести здесь текст патча не позволяет ограничение на ширину страницы.

```
#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;
    if (offset > 0) {
            *eof = 1;
        return 0;
    }
    len = sprintf(buffer, "count = %ld\n", value);
    return len;
}
void kill_count(struct siginfo *info, int pid, int sig)
{
    atomic_long_inc(&count);
}
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(kill_count);
error:
```

```
    return rv;
}
void hook_exit(void)
{
    set_kill_hook(NULL);
    remove_proc_entry(proc_entry, &proc_root);
    printk(KERN_INFO "%s was removed\n", proc_entry);
}
module_init(hook_init);
module_exit(hook_exit);
```

Очевидным расширениями модуля будет хранение в списке информации о том, как распределялся системный вызов по пользователям или по процессам и использование мьютекса для доступа к списку.

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

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

Заключение

В этом пособии рассмотрены лишь основы системного програмирования в ядре Linux. В качестве дальнейших шага можно предложить изучение [5]. Вопросы создания драйверов устройств, включая обработку прерывания, а также вопросы организации основных подсистем ядра, важных для понимания построения операционной системы, глубоко рассмотрены в [3, 6, 7].

Обсуждения вопросов создания ядра ведётся в списке рассылки Linux Kernel Mailing List [13]. В числе полезных интернет-ресурсов следует упомянуть `http://lwn.net/` и `http://kernelnewbies.org/`.

[13] Архивы доступны по адресу `http://lkml.org/`.

Список литературы

[1] Керниган Б., Ритчи Д. Язык программирования Си. – М: Вильямс, 2006. – 304 с.

[2] Таненбаум Э. С., Вудхалл А. Операционные системы. Разработка и реализация. – СПб: Питер, 2007 – 704 с.

[3] Corbet J., Rubini A., Kroah-Hartman G. Linux Device Drivers. – Sebastopol: O'Reilly, 2005. – 636 p.

[4] Kroah-Hartman G. Linux Kernel in a Nutshell. – Sebastopol: O'Reilly, 2006. – 198 p.

[5] Burian M. Salzman P. J. The Linux Kernel Module Programming Guide. – `http://tldp.org/LDP/lkmpg//`

[6] Бовет Д. , Чезати М. Ядро Linux. – М: BHV, 2007 – 1104 с.

[7] Лав Р. Разработка ядра Linux. – М: Вильямс, 2006 – 448 с.