Автор: Алексей Соловьев <>
Date: Mon, 10 Sep 2011 17:02:14 +0000 (UTC)
Subject: Разработка модулей ядра Linux

Оглавление

Введение

Самыми известными из всех операционных систем сейчас, несомненно, является семейство Windows корпорации Microsoft. Однако, несмотря на свою популярность, Windows не первая и не единственная операционная система в мире. Как известно, разработки операционных систем велись еще в 60-х годах XX века, но в силу специфики распространения компьютеров в то время, они не получили всеобщего распространения, оставаясь чисто академическими проектами. Первой системой, вышедшей за стены своего родителя, была Unix, сразу же завоевавшая место в образовательных учреждениях благодаря своей открытой природе. Впоследствии исходные тексты Unix стали коммерческой

тайной и были закрыты. Разработчики, желавшие непосредственно ознакомится с принципами функционирования операционных систем, вынуждены были довольствоваться одной теорией. Ситуация изменилась с появлением операционной системы Minix и ее наследницы Linux. Особенностью Minix и Linux являются открытые исходные тексты всей системы. Linux является второй по распространенности системой для компьютеров IBM PC и самой распространенной из открытых систем.

Тем не менее, в литературе по операционным системам почти не встречается конкретных описаний принципов работы отдельных подсистем, что значительно затрудняет освоение и совершенствование данной системы. Как отмечает создатель ОС Minix Эндрю Таненбаум, изучение одной только теории формирует у студентов однобокий взгляд на то, какой в действительности может быть операционная система. Действительно важные вещи зачастую опускаются, так как им не посвящено достаточно теории. [17]

Нужно отметить огромный интерес пользователей к принципам работы операционных систем. Так, вскоре после создания Minix, для обсуждения этой операционной системы была сформирована группа новостей USENET. За несколько недель на нее подписалось более сорока тысяч подписчиков, и большинство из них хотели добавить в систему множество новых возможностей, чтобы сделать ее лучше и больше. Каждый день несколько сотен человек давали советы, предлагали идеи и фрагменты кода. [17]

Почему же сейчас очень мало литературы, которая содержала бы описание исходных текстов и интерфейсов ядра Linux? Возможными причинами этого являются следующие:

1. Высокая коммерческая ценность технической документации.
   Так, в начале существования Unix ее исходные коды были широко доступны и часто изучались. Самой известной книгой такого рода была книга Lions' Commentary on Unix (опубликована впервые в 1977 году), написанная Джоном Лайонсом, описывающая шаг за шагом работу Unix. Эта книга использовалась во многих университетских курсах по операционным системам. Но затем стало ясно, что Unix превратилась в коммерческий продукт, поэтому было запрещено изучение исходного кода на учебных курсах, чтобы не подвергать риску его статус коммерческого секрета. [17]

   Как отмечает Николай Безруков, для того, чтобы создать предпосылки для успешного частного бизнеса, необходимо создать закрытую коммерческую инфраструктуру, обеспечивающую понимание кода, работающую только лично для компании. Скрытие информации об архитектуре может быть эффективной стратегией контроля над проектом с открытыми исходниками. [3]

2. Высокая сложность понимания исходных текстов.

Кривая затрат на изучение ядра современных ОС становится все длиннее и круче. Системы становятся все более сложными и, кроме того, очень большими по объему.

Понимание программ серьезная проблема. Она состоит в том, что не всегда достаточно иметь исходники. Если программа или система написаны на сравнительно низкоуровневом языке, вроде C, Cobol или Fortran, и плохо документирована, то все основные проектные решения растворились в деталях кодирования и требуют реконструкции. В таких ситуациях ценность документации более высокого уровня, такой, как спецификации интерфейсов и описание архитектуры, может превышать ценность самого исходного текста. [3]

Один из возможных подходов к решению данной проблемы был предложен создателем Linux Линусом Торвальдсом. Это ясность исходного кода: удобные интерфейсы, четкая структура, следование принципу "делать мало, но делать хорошо".
Подход, который предлагает Эндрю Мортон, состоит в использовании большего числа комментариев внутри исходного кода, что поможет читателю понять, чего хотел достичь программист.

Мы в своей работе будем придерживаемся подхода Роберта Лава, главного инженера по разработке ядра Linux корпорации Novell. Он состоит в предоставлении возможности, благодаря которой разработчики смогут получить полную информацию о том, какие задачи должны выполнять различные подсистемы ядра и каким образом предполагается выполнение этих задач.

Основная цель нашей работы предоставить материал, позволяющий начинающим разработчикам и студентам учиться и достигать уровня, которого нельзя достичь простым чтением кода ядра. Эта информация будет полезна для многих людей: для разработчиков прикладного ПО, для желающих ознакомиться с устройством ядра и других.

Для достижения данной цели необходимо решить следующие задачи:

1. Выделить основные подсистемы ядра.
2. Дать их функциональное описание.
3. Сформулировать практические задачи для приобретения навыков работы с подсистемой ядра.

Для возможности применения в учебном процессе работа состоит из двух частей: теоретической, в которой излагаются основные принципы функционирование подсистем ядра Linux. практической, где приводятся примеры программирования для рассмотренной подсистемы ядра, работа над которыми позволяет сформировать полное представление о работе

операционной системы.

Глава 1. Ядро Linux

1.1. Особенности ядра Linux

Linux своим появлением обязана в первую очередь операционной системе Unix, созданной подразделением Bell Laboratories компании AT&T. В 1969 году Дэннис Ритчи и Кен Томпсон создали операционную систему Unix для платформы PDP-7. В 1971 году операционная система Unix была перенесена на платформу PDP-11, а в 1973 году переписана с использованием языка C. Первая версия Unix, которая использовалась вне стен Bell Laboratories, называлась Unix System версии 6 (V6).

Другие компании переносили операционную систему Unix на новые типы машин. Эти версии содержали улучшения, которые позже привели к появлению нескольких разновидностей системы. В 1977 комбинация этих вариантов была выпущена в виде Unix System III, а в 1982 году корпорация AT&T представила версию System V.

С появлением Unix Version 7 она превратилась в коммерческий продукт и ее исходные коды перестали быть общедоступными. С другой стороны, на основе Unix Version 7 Эндрю Таненбаумом была создана операционная система Minix с открытым исходным кодом. Minix изначально задумывалась как обучающая система, пригодная для изучения студентами учебных заведений. Несмотря на многие предложения по улучшению системы, автор Minix не изменял этим целям.

Именно в этот момент Линус Торвальдс решил создать свой эмулятор терминала Minix. Этот эмулятор постепенно превратился в самостоятельную операционную систему, известную как Linux. После публикации исходного кода Linux в интернете к разработке системы подключились программисты со всего мира (прежде всего пользователи Minix). Стремление разработчиков Linux к расширению возможностей системы принесло результаты. Сейчас Linux это развитая операционная система, работающая на множестве платформ: Intel, Intel IA-64, AMD x86-64, PowerPC, Compaq Alpha, SPARC и других.

Операционная система Linux это клон Unix, в ней много заимствований из Unix, в ней реализован API ОС Unix (как это определено в стандарте POSIX),

но она не является производной от исходного кода Unix, как это имеет место в
других системах (например, *BSD). Там, где это было желательно, разработчики ядра Linux использовали свой подход, отличный от того, что
используется в Unix.

Ядро Linux, как и большая часть прикладных программ для системы Linux,
является программным обеспечением с открытым исходным кодом,
распространяемым по лицензии GPL. Строго говоря, полное название Linux операционная система GNU/Linux, что свидетельствует о присутствии в системе программ, написанных в рамках проекта GNU (GNU's Not Unix рекурсивная расшифровка). В дальнейшем термин Linux, если это не оговаривается заранее, будет применяться только к ядру. [10, 17]

1.1.1. Ядро Linux в сравнении с классическими ядрами Unix

Благодаря общему происхождению и одинаковому API, современные ядра Unix имеют много общих черт. За некоторым исключением это монолитные статические бинарные файлы. Они существуют в виде больших исполняемых образов, которые выполняются один раз и используют одну копию адресного
пространства. Для работы Unix обычно требуется система с контроллером управления страничной адресацией памяти (MMU, Memory Management Unit); эта система позволяет обеспечить защиту памяти в системе и предоставить
каждому процессу виртуальное адресное пространство. Ядро Linux не базируется на какой-то версии Unix, оно является монолитным, однако поддерживает некоторые свойства микроядерной архитектуры. Отличия Linux
от других разновидностей Unix: [10]

* Поддержка динамически загружаемых модулей ядра. Хотя ядро Linux является монолитным оно дополнительно поддерживает динамическую загрузку и выгрузку кода по мере необходимости. Эта возможность впервые появилась в версии 0.99 благодаря Питеру Мак-Дональду. [8] Часто в виде модулей поставляются драйверы устройств.

* Ядро Linux поддерживает симметричную многопроцессорную обработку (SMP). Большинство коммерческих вариантов Unix поддерживают SMP, но большинство традиционных реализаций этой поддержки не имеет.

* Ядро Linux является преемптивным. Ядро в состоянии вытеснить выполняющееся задание, даже если оно работает в режиме ядра. Среди коммерческих реализаций Unix преемптивное ядро имеют только Solaris,
IRIX, и сам Unix.

* В Linux использован иной подход к реализации потоков: потоки ничем не
отличаются от обычных процессов. С точки зрения ядра все процессы одинаковы, просто некоторые из них имеют общие ресурсы.

* В ядре Linux отсутствуют некоторые функции Unix, которые считаются плохо реализованными (например, STREAMS).

* Ядро Linux создается с использованием компилятора GNU C. Причем разработчики ядра используют расширение C ISO C99, поддерживающее функции с подстановкой тела (объявляются с помощью ключевых слов static inline), ассемблерные вставки (используется директива компилятора asm()), аннотацию ветвлений (макросы likely() и unlikely() более вероятное и менее вероятное события; используются для оптимизации кода) и другие возможности. [14]

* Ядро Linux полностью открыто. Заметим, что сборки ОС GNU/Linux (дистрибутивы) от разных производителей зачастую основываются на ядре, отличном от оригинального (канонического) ядра, выпускаемого разработчиками. Так, например, в дистрибутив Red Hat входит ядро с изменениями Алана Кокса, а дистрибутив Slackware использует каноническое ядро. [18] В качестве объекта нашей работы возьмем исходный код ядра Linux версии 2.6.18.

1.1.2. Ядро в сравнении с пользовательскими программами

Ядро операционной системы имеет некоторые специфические особенности в сравнении с программами, которые выполняются в пространстве пользователя. Вот некоторые из них: [10]

* Ядро не имеет доступа к стандартным библиотекам языка C. Причина этого скорость выполнения и объем кода. Даже самая необходимая часть библиотеки очень большая и неэффективная для ядра. Часть функций, однако, реализованы в ядре. Например, обычные функции работы со строками описаны в файле lib/string.c.

* Отсутствие защиты памяти. Если обычная программа предпринимает попытку некорректного обращения к памяти, ядро может аварийно завершить процесс. Если ядро предпримет попытку некорректного обращения к памяти, результаты могут быть менее контролируемыми. К тому же ядро не использует замещение страниц: каждый байт, используемый в ядре это один байт физической памяти.

* В ядре нельзя использовать вычисления с плавающей точкой. Активизация режима вычислений с плавающей точкой требует сохранения и проставления регистров устройства поддержки вычислений

с плавающей точкой, помимо других рутинных операций.

* Фиксированный стек. Стеком называют область адресного пространства, в которой выделяются локальные переменные. Локальные переменные это все переменные, объявленные внутри левой открывающей фигурной скобки тела функции (или любой другой левой фигурной скобки) и не имеющие ключевого слова static. [16] Стек в режиме ядра ни большой, ни изменяющийся. Поэтому в коде ядра не рекомендуется использовать рекурсию. Обычно стек равен двум страницам памяти, что соответствует 8 Кбайт для 32-разрядных систем и 16 Кбайт для 64 -разрядных.

* Переносимость. Платформо-независимый код, написанный на языке C, должен компилироваться без ошибок на максимально возможном количестве систем.

1.1.3. Дерево исходных кодов ядра

Дерево исходных кодов ядра содержит подкаталоги, описание которых приведено в таблице 1.1. В корне дерева также содержится ряд файлов: COPYING лицензия, CREDITS список основных разработчиков, MAINTAINERS список разработчиков, занимающихся поддержкой подсистем и драйверов ядра, Makefile основной сборочный файл ядра.

Таблица 1.1 Каталоги в корне исходного кода ядра Linux версии 2.6.18

Каталог	Описание
arch	Специфичный для аппаратной платформы код
block	Подсистема блочного ввода/вывода
crypto	Криптографический API
Documentation	Документация исходного кода
drivers	Драйверы устройств
fs	Подсистема VFS и отдельные файловые системы
include	Заголовочные файлы ядра
init	Загрузка и инициализация ядра
ipc	Код межпроцессорного взаимодействия
kernel	Основные подсистемы (планировщик и др.)
lib	Вспомогательные подпрограммы
mm	Подсистема управления памятью и поддержка виртуальной памяти
net	Сетевая подсистема
scripts	Сценарии компиляции кода
security	Модуль безопасности Linux
sound	Звуковая подсистема
usr	Начальный код пространства пользователя

1.1.4. Модули ядра

Несмотря на то, что ядро Linux является монолитным, оно позволяет выполнять динамическую вставку и удаление кода ядра в процессе работы. Загружаемый объект ядра называется модулем.

Модуль по своей сути примерно то же, что и обычная программа. Модуль так же имеет точку входа и выхода и находится в своем бинарном файле. Но
модули имеют непосредственный доступ к структурам и функциям ядра. Для программ в пространстве пользователя этот доступ ограничен библиотечными
интерфейсами компилятора.

Рассмотрим простейший модуль ядра (классический "Hello, world").

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
void sayHello()
{
    printk(KERN_INFO"Hello, world\n");
}
EXPORT_SYMBOL(sayHello);
static int __init hello_init(void)
{
    sayHello();
    return 0;
}
static void __exit hello_exit(void)
{
    printk(KERN_INFO"Goodbye, world\n");
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

Точка входа в модуль это функция, объявленная как static int __init.
Функция обязана в случае успешной загрузки возвращать нулевое значение.
Точка выхода функция, объявленная как static void __exit.

Основное назначение функции printk() дать ядру механизм регистрации
событий и предупреждений (в ядре версии 2.6.18 существует 8 уровней вывода
сообщений ядра, определенные в include/linux/kernel.h). Эти сообщения для
большинства систем можно посмотреть, обратившись к файлу /var/log/messages.
В данном случае, помимо этого, сообщение будет выведено на неграфический
терминал, с которого был загружен модуль.

Любой модуль ядра должен подключать заголовочный файл linux/module.h.

В файле linux/kernel.h содержатся определения макросов для функции printk(), в
том числе и KERN_INFO. В заголовочном файле linux/init.h содержатся определения макросов __init и __exit.

С помощью директивы EXPORT_SYMBOL() осуществляется экспорт функций ядра. Экспорт функций необходим, чтобы они были доступными для других модулей. Функция EXPORT_SYMBOL_GPL() также осуществляет экспорт в ядро, однако функция в этом случае будет доступна только модулям,
распространяющимся по лицензии GPL. В данном случае функция sayHello() будет доступна для других модулей. Директива MODULE_LICENSE() указывает на лицензию, под которой распространяется данный модуль. Некоторые функции могут быть доступны только для модулей,
распространяемых под лицензией GPL (например, функции для работы с очередями процессов). [2, 10]

1.2. Сборка

1.2.1. Сборка ядра

Далее в работе в описываемых примерах используется каноническое ядро
Linux версии 2.6.18.

Сборка ядра производится с помощью утилиты make.
После получения исходных кодов ядра в архиве linux-2.6.18.tar.bz2, его
нужно распаковать (обычно распаковывают в каталог /usr/src/) с помощью утилиты tar от имени суперпользователя (предположим, что архив находится в
текущем рабочем каталоге оболочки):

```
tar -C /usr/src/ -xjvf linux-2.6.18.tar.bz2
```

После распаковки переходим в появившийся каталог /usr/src/linux-2.6.18/:

```
cd /usr/src/linux-2.6.18/
```

Команда make defconfig создаст конфигурацию "по умолчанию" для текущей архитектуры. Обычно этого недостаточно, и ядро необходимо конфигурировать вручную. Вызов программы конфигурации ядра осуществляется командой make menuconfig. Дополнительную информацию о параметрах команды make можно найти в файле README.

Для того, чтобы ядро загрузилось в минимальной конфигурации, достаточно указать используемый драйвер контроллера IDE в разделе "Device
Drivers -> ATA/ATAPI/MFM/RLL support -> PCI IDE chipset support" (команда
make defconfig выбирает только VIA82CXXX). Также в разделе "File systems"
необходимо отметить используемые файловые системы (по умолчанию в ядре

присутствует поддержка только файловой системы ext2).

Сборка ядра осуществляется командой make. Ускорить процесс можно, запустив команду make -j2. Параметр -j2 означает запуск двух потоков выполнения (обычно используют два потока на процессор). Если некоторые компоненты ядра были выбраны как модули, то команда make modules_install
поместит их в каталог /lib/modules/2.6.18/kernel/.

```
make -j2
make modules_install
```

После сборки ядро находится по адресу arch/i386/boot/bzImage. Файл System.map это таблица, отображающая адреса символов ядра, находится в
корне дерева каталогов исходного кода ядра. [11] Эти файлы необходимо скопировать в каталог /boot/:

```
cp ./arch/i386/boot/bzImage /boot/vmlinuz-2.6.18
cp ./System.map /boot/System.map-2.6.18
```

Если создавался важный загрузочный драйвер (например, драйвер файловой
системы, который был собран в виде модуля), стартовый ram-диск позволит
загрузить его в процессе начальной загрузки. Стартовый ram-диск это некий
корневой псевдо-раздел, который живет в памяти и позже выполняет chroot на
реальный раздел диска (например, если корневой раздел расположен на RAID).
[11] Стартовый ram-диск создается при помощи команды mkinitrd. Параметры
запуска mkinitrd зависят от используемого дистрибутива. Для дистрибутивов
Fedora Core, Mandriva:

```
mkinitrd /boot/initrd-2.6.18 2.6.18
```

Для дистрибутива openSUSE (требуется указать файл System.map):

```
mkinitrd -k /boot/vmlinuz-2.6.18 -M /boot/System.map-2.6.18
-i /boot/initrd-2.6.18
```

Для остальных дистрибутивов необходимо предварительно ознакомиться со
справкой man mkinitrd.

После сборки необходимо сконфигурировать загрузчик (обычно LILO или GRUB) для доступа к соответствующему ядру.

Сборка ядра это первый шаг, который должен сделать любой будущий разработчик ядра.

Процесс сборки описан для ядер версии 2.6.x. Процесс для сборки ядер
версии 2.4.x отличается набором команд make. Поскольку ядра версии 2.4.x
сейчас используются реже версии 2.6.x, мы не будем их рассматривать.

Эксперименты с ядром могут быть пагубными для компьютера вплоть до полного краха системы и уничтожения всех данных на диске. Поэтому в целях безопасности для экспериментов рекомендуется использовать предустановленный на виртуальную машину дистрибутив. Помимо безопасности, таким образом обеспечивается единая конфигурация используемых компьютеров. В качестве дистрибутива рекомендуется использовать Slackware Linux, поскольку в нем используются не модифицированные программные компоненты. В качестве виртуальной машины можно (хотя не обязательно) использовать виртуальную машину QEMU или VMware. Виртуальный жесткий диск при установке системы рекомендуется разбивать на два раздела: раздел подкачки (/dev/hda1), и корневой (/dev/hda2), отформатированный в файловой системе ext2 (самая простая схема разбиения). В этом случае на этапе сборки ядра для QEMU (и VMware)
необходимо указать используемый контроллер IDE: "Device Drivers -> ATA/ATAPI/MFM/RLL support -> PCI IDE chipset support -> Intel PIIXn chipset support".

Сборка ядра занимает довольно продолжительное время около 20 минут.
Для ядра, собранного в конфигурации "по умолчанию", можно не создавать ram
диск, поскольку никаких важных загрузочных драйверов не создается.

В качестве текстового редактора для конфигурационных файлов загрузчика
рекомендуется использовать редактор mcedit как наиболее простой в освоении.
В Slackware Linux по умолчанию используется системный загрузчик LILO.

Конфигурационный файл LILO /etc/lilo.conf. Команда mcedit /etc/lilo.conf
откроет файл для редактирования. В конец файла добавляются строки:

```
image = /boot/vmlinuz-2.6.18
root = /dev/hda2
label = linux-2.6.18
read-only
```

Если в конфигурационном файле есть параметр vga, он должен быть установлен в значение normal. Таким образом ядро будет загружать без использования режима framebuffer. Поскольку в конфигурацию по умолчанию
поддержка framebuffer не входит, иное значение параметра vga может вызвать
сбой загрузки.

```
vga = normal
```

Установка загрузчика LILO осуществляется командой /sbin/lilo.

1.2.2. Сборка модулей ядра

Сборка модуля также производится с помощью утилиты make. Для этого необходимо создать Makefile. Предположим, что описанный выше модуль
сохранен в файле с именем hello.c. Тогда содержимое Makefile будет:

```
obj-m += hello.o
```

Makefile и файл hello.c размещаются в одном каталоге. Затем из этого
каталога дается команда:

```
make -C /usr/src/linux-2.6.18 SUBDIRS=$PWD modules
```

Для сборки собственного модуля нам потребуется исходный код ядра (обычно он хранится в каталоге /usr/src/). Параметр -C говорит утилите make,
что перед совершением каких-либо действий необходимо сменить рабочий каталог на указанный (мы должны указать путь к каталогу исходных кодов ядра). Если команда make будет выполнена успешно, то в каталоге модуля появится файл hello.ko.

После сборки модуль можно загрузить командой insmod <имя модуля> (в нашем случае это insmod hello.ko). Список загруженных модулей можно посмотреть в /proc/modules. Выгрузка модуля осуществляется командой rmmod
<имя модуля> (rmmod hello.ko). Для этих команд необходимы права суперпользователя.

1.2.3. Сборка модулей вместе с ядром

Linux открытая система и ядро можно скомпилировать с написанным модулем при помощи системы сборки "kbuild". Пусть, например, мы хотим разместить наш модуль в разделе "Kernel hacking". Настройки параметров сборки ядра редактируются с помощью файлов Kconfig*. В нашем случае необходимо добавить в файле arch/i386/Kconfig.debug такие строки:

```
config MYKERNEL
  bool "My kernel modules"
  default y
  help
  This is my kernel modules
if MYKERNEL
config MYKERNEL_SAYHELLO
   bool "sayHello() function"
   default y
   help
   Simple function sayHello() that say "Hello, world".
endif
```

В меню "Kernel hacking" теперь станет доступен дополнительный параметр
CONFIG_MYKERNEL (в файлах Kconfig* не пишется префикс CONFIG_).
Если он выбран, то становятся доступными параметры, указанные после директивы if MYKERNEL. Конфигурация ядра хранится в файле .config.

Тип bool означает, что параметр может иметь только два значения: Y/N. Тип
tristate означает, что параметр может иметь три значения: Y/N/M. С помощью
директивы default можно задать значение по умолчанию для модуля.
Расшифровка значений:

Y модуль включается в бинарный файл ядра на этапе сборки. В этом случае важно, чтобы функции, обозначающие точки входа и выхода, имели уникальное имя, то есть не перекрывались в пространстве имен ядра.

M данный модуль должен быть скомпилирован как динамически подгружаемый модуль ядра.

N модуль не будет компилироваться.

Однако, чтобы модуль был скомпилирован, нам необходимо разместить исходный код модуля в одном из каталогов дерева исходных кодов ядра. Поместим код модуля (файл hello.c) вместе с Makefile в каталог kernel/hello/
дерева исходных кодов ядра. В файл kernel/Makefile добавим строку:

```
obj-$(CONFIG_MYKERNEL_SAYHELLO) += hello/
```

Если для модуля hello выбрано значение Y или M, то в сборке ядра будет
участвовать переменная CONFIG_MYKERNEL_SAYHELLO, имеющая
значение Y или M соответственно. В случае, если значение параметра равно M,
файл hello.ko будет помещен командой make modules_install в одном из подкаталогов в каталоге /lib/modules/2.6.18/kernel/. В нашем случае, поскольку
модуль был добавлен в Makefile каталога kernel/, модуль будет размещен в
подкаталоге kernel/hello/ и полный путь будет выглядеть так:

```
/lib/modules/2.6.18/kerne/kernel/hello/hello.ko
```

Ядро в различных дистрибутивах имеет дополнительный суффикс в названии версии. Например, ядро дистрибутива Fedora Core 6, хотя и основано
основано на ядре Linux версии 2.6.18, но имеет свою версию 2.6.18-1.2798.fc6
значение дополнительного суффикса записывается в переменной
EXTRAVERSION Makefile ядра. Например, для Fedora Core 6:

```
EXTRAVERSION = -1.2798.fc6
```

Изменив значение переменной EXTRAVERSION, можно дать уникальное имя модифицированному ядру. Например:

```
EXTRAVERSION = mykernel
```

Для самопроверки модуль можно собирать именно как модуль. Тогда парамтр CONFIG_HELLO должен иметь тип tristate. В конфигурационной записи следует писать:

```
tristate "sayHello() function"
default m
```

Не обязательно помещать текст в файл arch/i386/Kconfig.debug. Можно нример, создать файл kernel/hello/Kconfig и поместить конфигурационную запись туда. В файл arch/i386/Kconfig.debug в таком случае нужно добавить строку:

```
source "kernel/hello/Kconfig"
```

Обратите внимание, что относительный путь в файлах Kconfig* записывается от корневого каталога дерева исходных кодов. В отличие от Makefile'ов, в которых относительный путь записывается от текущего файла Makefile.

Если изменить переменную EXTRAVERSION, как показано на примере в тексте, то команда uname -r (версия ядра) вернет 2.6.18-mykernel.

1.2.4. Утилиты diff и patch

Все изменения исходного кода ядра распространяются в виде заплат (patch). Заплаты это результат вывода утилиты diff в формате, который подается на вход утилиты patch. Проще всего заплату сгенерировать, когда имеется два дерева исходных кодов ядра: одно оригинальное, другое дерево с изменениями. Предположим, оба дерева находятся в одном каталоге /usr/src/. Для генерации заплаты необходимо выполнить команду diff из этого каталога:

```
diff -ruN linux-2.6.18/ linux-2.6.18-2/ > mykernel.patch
```

Параметр -r означает, что анализ каталогов необходимо проводить

рекурсивно. Параметр -u означает, что необходимо использовать
унифицированный формат вывода утилиты diff. Применяется для
удобочитаемости заплаты. Параметр -N означает, что новые файлы, которые
появились в измененном каталоге, дролжны быть включены в заплату. Для одного файла формат команды diff будет следующим:

```
diff -u linux-2.6.18/file linux-2.6.18-2/file > file.patch
```

Утилиту diff следует запускать из каталога, в котором находятся оба дерева
исходных кодов. Это принятое правило генерации заплат. Для того, чтобы применить заплату, необходимо выполнить команду patch из корневого каталога дерева исходных кодов оригинального ядра (в нашем случае это /usr/src/linux-2.6.18/):

```
patch -p1 < /usr/src/mykernel.patch
```

Чаще всего заплаты помещают в каталог /usr/src/. Параметр -p1 означает,
что необходимо игнорировать имя первого каталога в путях всех файлов, в
которые будут вноситься изменения. Это позволяет применить заплату независимо от того, какие имена каталогов кода ядра были машине, где создавалась заплата.

Утилита diffstat позволяет сгенерировать гистограмму изменений, к которым приведет применение заплаты (удаление и добавление строк). Для этого необходимо выполнить команду:

```
diffstat -p1 /usr/src/mykernel.patch
```

Заплаты отправляют в список рассылки разработчиков ядра (Linux Kernel
Mailing List, lkml). Поскольку утилита patch игнорирует все строки до тех пор,
пока не встретит формат diff, то вначале заплаты можно включить ее описание.

Заплата для генерации модифицированного ядра, полученного в ходе работы, находится в файле mykernel.patch приложений. Параметры
конфигурации модулей, описанных в работе, находится в разделе "Kernel hacking". Сразу заметим, что мы не можем протестировать изменения на аппаратных платформах, отличных от i386, таких как Alpha или SPARC.

Поэтому модифицированное ядро было протестировано только на платформе i386. Чтобы модифицированное ядро было успешно скомпилировано,
необходимо в меню конфигурации ядра установить параметры "Processor type
and features -> Processor family -> 386" и "Processor type and features ->

Subarchitecture type -> PC-compatible".

Глава 2. Разработка ядра Linux

2.1. Системные вызовы

2.1.1. Системные вызовы в Linux

Ядро предоставляет набор интерфейсов, именуемых системными вызовами, которые обеспечивают взаимодействие прикладных программ, работающих в пространстве пользователя, и аппаратной части компьютера. Например, при работе с файлами программы могут не заботиться о типе жесткого диска и файловой системе на нем.

Системные вызовы гарантируют безопасность и стабильность системы. Так как ядро работает посредником между ресурсами системы и программами, оно может принимать решения о предоставлении доступа в соответствии с правами пользователя и другими критериями.

Прикладные программы разрабатываются с применением программных интерфейсов приложений (Application Programming Interface, API). В этом случае нет необходимости между в корреляции между интерфейсами, которые используют приложения и интерфейсами, которые предоставляет ядро. POSIX, WinAPI примеры таких API. Может существовать один и тот же API для различных операционных систем, а реализация его может отличаться. Например, операционные системы Linux и FreeBSD соответствуют стандарту POSIX (Linux на 100% соответствует стандарту POSIX 1003.1 [9]), и многие приложения, написанные для FreeBSD, могут сравнительно легко быть перенесены в Linux и наоборот. Этим объясняется схожесть наборов приложений для этих операционных систем.

Частично интерфейс к системным функциям обеспечивает библиотека C. Например, функция printf() формирует строку в соответствии с заданным форматом и передает ее системному вызову write(), который отправляет ее на стандартное устройство вывода (чаще всего терминал).

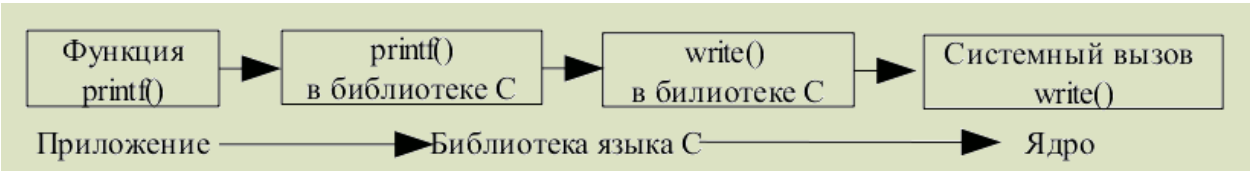

Рис. 2.1. Взаимодействие между приложением, библиотекой C и ядром на примере функции printf()

Дополнительно библиотека функций языка C предоставляет большую часть
API-стандарта POSIX. С помощью команды strace можно отслеживать обращения программы к ядру, производимые с помощью системных вызовов (поэтому ее можно использовать для отладки и поиска причины ошибки). [10]

Реализация системных вызовов

Системные вызовы (в ОС Linux их часто именуют syscall) обычно реализуются в виде функции с возвращаемым значением типа long. Для них могут быть определены один или боле аргументов. В случае ошибки системные
вызовы записывают специальный код ошибки в глобальную переменную errno.
Значение errno может быть переведено с помощью библиотечной функции perror(), которая выводит в стандартный поток сообщения, описывая ошибку,
произошедшую при последнем системном вызове или вызове библиотечной функции.

Рассмотрим системный вызов getuid() (реализован в kernel/timer.c), который
возвращает uid текущего процесса:

```
asmlinkage long sys_getuid(void)
{
    return current->uid;
}
```

Модификатор asmlinkage говорит компилятору, что обращение к функции должно производиться только через стек. Все системные вызовы возвращают
значение типа long. Системный вызов getuid() объявлен как sys_getuid(). Это
соглашение о присваивании имен, принятое в системе Linux.

Каждому системному вызову в Linux присвоен уникальный номер системного вызова (syscall number). Процессы не обращаются к системным вызовам по имени, вместо этого они используют его номер. Однажды назначенный номер не должен меняться никогда для совместимости с прикладными программами. Если системный вызов удаляется,
соответствующий номер не должен использоваться повторно. В ядре Linux версии 2.6.18 реализовано 318 системных вызовов для платформы i386 (это
значение хранится в постоянной NR_syscalls). Объявление вызовов находится в
файле include/asm/unistd.h. Этому файлу в коде ядра соответствует файл include/asm-i386/unistd.h в дереве исходных кодов ядра. Для лучшей переносимости в коде ядра вместо asm-<тип архитектуры> используется asm
(на этапе сборки создается символическая ссылка include/asm на каталог include/asm-i386/). Поэтому в дальнейшем под каталогом asm/ будет подразумеваться каталог asm-i386/.

В Linux предусмотрен "не реализованный" ("not implemented") системный вызов функция sys_ni_syscall(), которая не делает ничего, кроме того, что возвращает значение, равное -ENOSYS код ошибки, соответствующий неправильному системному вызову. Эта функция используется вместо удаленных системных вызовов.

Ядро хранит список зарегистрированных системных вызовов в таблице системных вызовов (syscall table). Таблица хранится в памяти, на которую указывает переменная sys_call_table. Данная таблица для платформы i386 (каждая платформа позволяет определять свои уникальные системные вызовы) хранится в файле arch/i386/kernel/syscall_table.S. [10]

Обработка системных вызовов

Пользовательские программы, если хотят обратиться к ядру через системный вызов, должны использовать определенный механизм. Таким механизмом является программное прерывание: создается исключительная ситуация (exception), и система переключается в режим ядра для выполнения обработчика этой ситуации. В данном случае это обработчик системного вызова (system call heandler) зависимая от аппаратной платформы функция system_call() (функция на языке ассемблера, реализована в arch/i386/kernel/entry.S).

Многие системные вызовы осуществляют переход в режим ядра одинаковым образом, поэтому ядру должен также передаваться номер системного вызова. Для аппаратной платформы i386 этот номер сохраняется в регистре eax. Обработчик системных вызовов считывает из регистра eax это значение. Функция system_call() проверяет правильность системного вызова сравнением с постоянной NR_syscalls. если значение больше или равно NR_syscalls, возвращается значение -ENOSYS, в противном случае вызывается соответствующий системный вызов:

```
call *sys_call_table(,%eax,4)
```

Большинство системных вызовов требуют также передачи одного или нескольких параметров. Самый простой способ передачи параметров это сохранить их в регистрах процессора. Для платформы i386 регистры ebx, ecx, edx, esi, edi хранят соответственно первые пять аргументов. В случае шести или более аргументов используется регистр, который содержит указатель на память пространства пользователя, где хранятся все параметры.

Возвращаемое значение также передается через регистр. Для платформы i386 оно сохраняется в регистре eax. [10]

2.1.2. Создание нового системного вызова

Задание 1

Добавить новый системный вызов в ядро. Системный вызов должен возвращать размер стека ядра.

Ход работы

Значение размера стека ядра в постоянной THREAD_SIZE, определенной в файле include/asm/thread_info.h. Системный вызов возвращает это значение.

```
#include <asm/thread_info.h>
asmlinkage long sys_getstsize(void)
{
    return THREAD_SIZE;
}
```

Регистрация нового системного вызова проходит в несколько этапов.

Код системного вызова размещается в дереве исходных кодов ядра. Разместим код системного вызова в файле arch/i386/kernel/new_calls.c. Вообще, лучше размещать код в наиболее подходящем файле. Для того, чтобы файл new_calls.c был скомпилирован с ядром, необходимо вписать в файл arch/i386/kernel/Makefile строку

```
obj-y += new_calls.o
```

Добавляется новая запись в конец таблицы системных вызовов в файле arch/i386/kernel/syscall_table.S. Это необходимо сделать для всех платформ, которые поддерживают данный системный вызов (как в нашем случае, но мы ограничимся только платформой i386). Отсчет в таблице начинается с нуля.

```
.long sys_tee             /* 315 */
.long sys_vmsplice
.long sys_move_pages
.long sys_ getstsize
```

Для всех поддерживаемых платформ номер системного вызова должен быть определен в файле unistd.h (мы ограничимся только файлом include/asm/unistd.h).

```
#define __NR_tee                    315
#define __NR_vmsplice               316
#define __NR_move_pages             317
#define __NR_ getstsize             318
```

Значение постоянной NR_syscalls тоже должно быть изменено (увеличено
на количество добавляемых вызовов в нашем случае 1).

```
#define NR_syscalls 319
```

После этого ядро собирается с новым системным вызовом.

Непрямой доступ к вызовам

Новый системный вызов не поддерживается библиотекой языка C. Но в ОС
Linux существует функция syscall(), предоставляющая непрямой доступ к системным вызовам. В качестве параметра в функцию syscall() передаются номер системного вызова и далее его параметры в том же порядке, в котором
они определены в коде ядра. Функция syscall() впервые появилась в ОС 4.0BSD
(информация взята из справки man 2 syscall). Для работы с функцией необходимо подключить заголовочный файл sys/syscall.h.

```
#include <sys/syscall.h>
```

Например, рассмотрим системный вызов open(), который определен следующим образом.

```
long open(const char *filename, int flags, int mode)
```

Вызов этой функции с помощью syscall() будет выглядеть так.

```
#define __NR_open 5
syscall(__NR_open, filename, flags, mode)
```

Постоянная __NR_open это номер системного вызова, определенный в файле include/asm/unistd.h.

Задание 2

Реализовать доступ к новому вызову. Для этого написать программу, эксплуатирующую новый системный вызов.

Ход работы

Исходный код программы на языке C:

```
#include <sys/syscall.h>
```

```
#include <stdio.h>
#define __NR_getstsize 318
int main()
{
    long stsize;
    stsize = syscall(__NR_getstsize);
    printf("Kernel stack size = %ld bytes\n", stsize);
    return 0;
}
```

В переменной stsize типа long сохраним результат системного вызова getstsize() и затем выведем сообщение с размером стека. После компиляции и
запуска программы, получаем сообщение, которое подтверждает, что новый системный вызов работает.

```
# gcc getstsize.c -o getstsize; ./getstsize
Kernel stack size = 8192 bytes
```

Код программы находится в файле getstsize.c приложений. Код системного
вызова находится в файле new_calls.c приложений.

2.2. Управление памятью в Linux

2.2.1. Страницы памяти

Ядро рассматривает страницы физической памяти как основные единицы управления памятью. Наименьшая единица памяти, которую может адресовать
процессор это машинное слово, однако модуль управления памятью (MMU) обычно работает со страницами памяти. Модуль MMU управляет таблицами страниц на уровне страничной детализации. Для каждой аппаратной платформы существует свой объем страниц памяти. Большинство 32-разрядных
платформ имеют размер 4 Кбайт, а большинство 64-разрядных 8 Кбайт. Таким образом, на 32-разрядной платформе объем памяти в 1 Гбайт разбивается на 262144 страницы.

Структура page

Страница представлена в ядре в виде структуры page, описанной в файле
include/linux/mm.h. Важно понимать, что структура page описывает страницы
физической, а не виртуальной памяти. Ядро использует эту структуру, чтобы
описывать область физической памяти, а не данных, которые в ней содержаться. Размер структуры равен 40 байт. В этом случае для описания всех
страниц памяти объемом 1 Гбайт используется 262.144 * 40 = 10.485.760 байт,
то есть 10 Мбайт. [10]

Зоны памяти

Из-за ограничений аппаратного обеспечения ядро не может рассматривать
все страницы физической памяти как идентичные. Некоторые страницы не могут использоваться для некоторых типов задач. Поэтому ядро делит всю доступную физическую память на зоны. В зонах представлены страницы с аналогичными свойствами. Ядро должно учитывать ограничения аппаратного обеспечения, связанные с адресацией памяти:

* Некоторые устройства могут выполнять прямой доступ к памяти (DMA,
Direct Memory Access) только в определенную область.

* На некоторых платформах для физической адресации доступны большие
объемы, чем для виртуальной. Поэтому часть памяти не может постоянно
отображаться в ядро.

На основании этих ограничений ядро разделяет всю память на три зоны:

ZONE_DMA. Страницы, совместимые с режимом DMA.

ZONE_NORMAL. Страницы, которые отображаются в адресные
пространства пользователя обычным способом.

ZONE_HIGHMEM. "Верхняя память", содержащая страницы, которые не могут постоянно отображаться в адресное пространство ядра.

Размер зон сильно зависит от типа процессора. Для платформы i386 размер
ZONE_DMA равен 16 Мбайт, ZONE_HIGHMEM это все, что лежит выше отметки 896 Мбайт, ZONE_NORMAL вся остальная память (16-896 Мбайт). Память в зоне ZONE_HIGHMEM называется "верхней памятью" (high memory), вся остальная память называется "нижней памятью"(low memory). Зона памяти
это логическое группирование, и оно никак не связано с аппаратным обеспечением. Каждая зона представлена в include/linux/mmzone.h структурой
zone. В структуре представлены количество свободных страниц в зоне (unsigned
long free_pages), имя зоны (char *name, возможные значения: "DMA", "Normal",
"HighMem") и другие значения. [10]

2.2.2. Интерфейсы для работы с памятью

Функции kmalloc() и kfree()

Функция kmalloc() (объявлена в include/linux/slab.h) аналогична malloc()
пространства пользователя за исключением добавленного параметра flags. kmalloc() выделяет участок памяти с заданным размером байт. Выделенные

страницы памяти являются виртуально смежными.

```
void *kmalloc(size_t size, gfp_t flags)
```

Функция возвращает указатель на область памяти размером не меньше size
байт. В случае ошибки возвращается значение NULL.

Функция kfree() (объявлена в include/linux/slab.h) позволяет освободить
память, ранее выделенную kmalloc() (вызывать kfree() надо для участков,
которые предварительно были выделены kmalloc()).

```
void kfree(const void *);
```

Вызов kfree(NULL) специально проверяется и поэтому является
безопасным. Пример использования функций:

```
struct some * p;
p = kmalloc(sizeof(some), GFP_KERNEL);
if (!p)
   /*обработчик ошибки*/
else {
    /*что-то делаем*/
    kfree(p);
}
```

Функции vmalloc() и vfree()

Функция vmalloc() объявлена в include/linux/vmalloc.h. Функция возвращает
указатель на виртуально непрерывную область размером не менее size байт. В
противном случае функция возвращает NULL.

```
void *vmalloc(unsigned long size)
```

Функция работает аналогично kmalloc(), но выделяет страницы, которые
виртуально смежные, но необязательно смежные физически. Однако функция
vmalloc() менее производительна по сравнению с kmalloc(), поскольку
страницы, выделенные vmalloc(), должны отображаться посредством таблиц
страниц. Это приводит к менее эффективному использованию буфера TLB (см.
раздел "адресное пространство процесса"). Функция vmalloc() используется для
выделения очень больших участков памяти. Например, при динамической
загрузке модулей ядра память для модуля выделяется с помощью vmalloc().

Для освобождения памяти, выделенной ранее функцией vmalloc(),

используется функция vfree() (объявлена в include/linux/vmalloc.h):

```
void vfree(void *addr)
```

Функция освобождает участок памяти, начинающийся с адреса addr, выделенный ранее функцией vmalloc(). Ничего не возвращает.

Флаги gfp_mask

Флаги модифицируют работу подсистемы памяти в зависимости от ситуации и разбиты на три категории:

* модификаторы операций. Указывают, каким образом ядро должно использовать память.
* модификаторы зон. Указывают, откуда ядро должно выделять память.

* флаги типов. Различные комбинации первых двух категорий.

Все флаги определены в файле include/linux/gfp.h.

Таблица 2.1 Модификаторы операций выделения памяти

Флаг	Описание
__GFP_WAIT	Операция не может переводить текущий процесс в состояние ожидания
__GFP_HIGH	Операция может обращаться к аварийным запасам
__GFP_IO	Операция может использовать дисковые операции ввода/вывода
__GFP_FS	Операция может использовать операции ввода/вывода файловой системы
__GFP_COLD	Операция должна использовать страницы, содержимое которых не находится в кэше процессора (cache cold)
__GFP_NOWARN	Операция не будет печатать сообщение об ошибках
__GFP_REPEAT	Операция повторит попытку в случае ошибки
__GFP_NOFAIL	Операция будет повторять попытки выделения неограниченное число раз
__GFP_NORETRY	Операция никогда не будет повторять попытку
__GFP_COMP	Добавить метаданные составной (compound) страницы памяти. Используется для поддержки больших страниц памяти
__GFP_ZERO	В случае успеха, вернуть страницу, заполненную нулями

Таблица 2.2 Модификаторы зоны

Флаг	Описание
__GFP_DMA	Выделять память только из зоны ZONE_DMA
__GFP_HIGHMEM	Выделять память только из зон ZONE_HIGHMEM и ZONE_NORMAL

Таблица 2.3 Флаги типов

Флаг	Описание

GFP_ATOMIC Запрос высокоприоритетный и в состояние ожидания переходить нельзя
GFP_NOIO Запрос может блокироваться, но при его выполнении нельзя выполнять операции дискового ввода/вывода
GFP_NOFS Запрос на выделение памяти может блокироваться и выполнять дисковые операции ввода/вывода, но запрещено выполнять операции, связанные с файловой системой
GFP_KERNEL Обычный запрос на выделение, который может блокироваться. Флаг предназначен для использования в коде, который выполняется в контексте процесса
GFP_USER Обычный запрос на выделение, который может блокироваться. Флаг используется для выделения памяти процессам пространства пользователя
GFP_HIGHUSER Запрос на выделение памяти из зоны ZONE_HIGHMEM, который может блокироваться
GFP_DMA Запрос на выделение памяти из зоны ZONE_DMA

Таблица 2.4 Соответствие флагов типов и модификаторов

Флаг	Модификаторы
GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_WAIT
GFP_NOFS	__GFP_WAIT | __GFP_IO
GFP_KERNEL	__GFP_WAIT | __GFP_IO | __GFP_FS
GFP_USER	__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL
GFP_HIGHUSER	__GFP_WAIT __GFP_IO | __GFP_FS | __GFP_HARDWALL | __GFP_HIGHMEM
GFP_DMA	__GFP_DMA

Большинство операций выделения памяти в ядре используют флаг GFP_KERNEL. Операции имеют обычный приоритет. При использовании флага GFP_KERNEL ядро попытается вытеснить страницы в swap.

Выделение памяти с флагом GFP_NOIO не будет запускать операций ввода/вывода. С флагом GFP_NOFS могут запускаться операции ввода/вывода,
но не могут запускаться операции файловых систем. Эти флаги используются в
основном в коде файловых систем или в коде низкоуровневого ввода/вывода.

Флаг GFP_DMA указывает, что память обязательно должна быть выделена из зоны ZONE_DMA. Используется в основном драйверами устройств. [10]

2.2.3. Работа с памятью в ядре

Задание 1

Создать системный вызов upper(), который будет преобразовывать символы
строки, хранящейся в адресном пространстве пользователя, к верхнему регистру.

Ход работы

Системный вызов будет принимать три параметра: указатель на строку,

которую необходимо преобразовать, указатель на область памяти, куда необходимо поместить результат, и длину этой строки.

```
asmlinkage long sys_upper(char *src, char *dst, int len)
{
    int i;
    char *buff;
    buff = (char *)kmalloc(len, GFP_KERNEL);
    memset(buff, 0, len);
    copy_from_user(buff, src, len);
    for (i = 0; i < len; i++)
          if((buff[i] >= 0x61) && (buff[i] <= 0x7A))
buff[i] -= 0x20;
    copy_to_user(dst, buff, len);
    kfree(buff);
    return len;
}
```

Для новой строки необходимо выделить область памяти в адресном пространстве ядра с помощью функции kmalloc(). В функцию передается флаг GFP_KERNEL. Функция memset() заполняет область памяти размером len байт значением 0, начиная с позиции, на которую указывает указатель buff.

Функция copy_from_user() копирует исходную строку из адресного пространства пользователя в адресное пространство ядра. Поскольку строка находится в пользовательском адресном пространстве, здесь нельзя использовать простое присвоение (иначе это создаст угрозу безопасности).

Затем производится смена регистра символов, находящихся в диапазоне 97 122 (0x61 0x7A в шестнадцатеричной системе счисления): английские символы нижнего регистра.

Результат (преобразованная строка) копируется из адресного пространства ядра в пространство пользователя при помощи функции copy_to_user(). Функция принимает в качестве параметров указатель выходную строку (в пространстве ядра), указатель на входную строку (в пространстве пользователя) и длину входной строки.

Объявление функций copy_from_user и copy_to_user() находится в файле include/asm-i386/uacces.h. Однако при подключении заголовочного файла в таком случае тип архитектуры не пишется (пишется просто "asm"). Объяснение этому было дано выше.

```
#include <asm/uacces.h>
```

После преобразований область памяти необходимо освободить с помощью функции kfree().

Поместим реализацию нового системного вызова в файл arch/i386/kernel/new_calls.c. Процесс регистрации нового системного вызова
был описан в параграфе 2.1 "Системные вызовы".

Задание 2

Написать программу, которая будет принимать строку символов в качестве
аргумента и возвращать преобразованную с помощью нового системного вызова upper() строку.

Ход работы

Исходный код программы на языке C:

```
#include <sys/syscall.h>
#include <stdio.h>
#define __NR_upper 319
int main(int argc, char *argv[])
{
    char *s1, *s2;
    int len, ret;
    if (argc > 1)
           s1 = argv[1];
    else
           s1 = "This Is (1234) Default TEST String";
    len = strlen(s1);
    s2 = (char *)malloc(len);
    memset(s2, 0, len);
    printf("Input string: %s\n", s1);
    ret = syscall(__NR_upper, s1, s2, len);
    printf("Output string: %s\n", s2);
    return 0;
 }
```

Программа получает в качестве параметра исходную строку s1. Если никакой строки в качестве параметра не передается (параметр argc равен 1), то
строка s1 инициализируется строкой по умолчанию, содержащей символы верхнего, нижнего регистров, символы арабских цифр и символы скобок ("This
Is (1234) Default TEST String"). Программа на выходе печатает исходную строку
s1 и конечную строку s2, полученную из исходной преобразованием с помощью системного вызова upper().

```
# gcc upper.c -o upper; ./upper
Input string: This Is (1234) Default TEST String
Output string: THIS IS (1234) DEFAULT TEST STRING
# ./upper HeLlo
Input string: HeLlo
Output string: HELLO
```

Код программы находится в файле upper.c приложений. Код системного вызова находится в файле new_calls.c приложений.

2.3. Процессы

2.3.1. Процессы в Linux

В Linux используется уникальная реализация потоков: между процессами и потоками нет никакой разницы. Многопоточность организована в виде процессов с общими ресурсами. Иное название для процесса задание или задача (task). Существуют также потоки в пространстве ядра (kernel thread) процессы, выполняемые строго в пространстве ядра. Тем не мене они планируются и выполняются как обычные процессы. По возможности, о программах, работающих в ядре говорят как о задачах, для работающих в режиме пользователя используют термин процесс.

В современных операционных системах процессы предусматривают наличие двух виртуальных ресурсов: виртуального процессора и виртуальной памяти. Виртуальный процессор создает иллюзию того, что процесс использует всю компьютерную систему, даже если физическим процессором пользуются другие процессы. Виртуальная память создает иллюзию того, что процесс использует всю доступную физическую память. Потоки в Linux совместно используют одну и ту же виртуальную память, хотя каждый поток получает свой виртуальный процессор.

Процесс начинает свое существование с момента создания. Создание процесса в операционной системе Linux (и во всех Unix системах) выполняется с помощью системного вызова fork() (буквально ветвление), который создает новый процесс путем копирования уже существующего. Процесс, вызывающий fork(), называется порождающим (родительским, parent), новый процесс называют порожденным (дочерним, child). После вызова fork() родительский процесс продолжает выполнение, а порожденный процесс выполняется с места возврата из системного вызова. Часто после ветвления в одном из процессов необходимо выполнить какую-то программу.
Семейство вызовов exec*() позволяет создать новое адресное пространство и загрузить в него новую программу. Выход из программы осуществляется с помощью системного вызова exit(), который завершает процесс и освобождает все занятые им ресурсы. Завершенный процесс переходит в состояние зомби, которое используется для представления завершенного процесса до момента, пока порождающий процесс не удалит его.

Ядро хранит информацию о всех процессах в двухсвязном кольцевом списке, называемом списком задач (task list). Элементами списка являются структуры task_struct, определенные в файле include/linux/sched.h, и

называемые дескриптором процесса. Структура занимает около 1,7 Кбайт на
32-разрядных системах, однако она полностью описывает процесс.

Система идентифицирует процессы с помощью уникального значения pid, называемое идентификатором процесса. Для совместимости со старыми версиями Linux и Unix максимальное значение по умолчанию равно 32768 (это
значение хранится в /proc/sys/kernel/pid_max).

В Linux существует четкая иерархия процессов. Все процессы являются потомками процесса init (pid = 1). Каждый процесс в системе имеет только
одного родителя. Процессы, имеющие общего родителя, называются
родственными (sibling). Структура task_struct содержит указатель на
дескриптор родительского процесса (поле struct task_struct *parent) и список
порожденных процессов children (поле struct list_head children). Доступ к
текущему процессу осуществляется с помощью макроса current.

Системный вызов getpid() возвращает pid текущего процесса:

```
asmlinkage long sys_getpid(void)
{
    return current->tgid;
}
```

Почему возвращается значение tgid (идентификатор группы потоков)? Для
обычных процессов значение tgid совпадает со значением pid. Но поскольку
процессы в Linux не отличаются от потоков, то текущим процессом может оказаться на самом деле порожденный поток. При наличии нескольких потоков
значение tgid одинаково для всех потоков одной группы. Такая реализация дает
возможность получать одинаковое значение для процессов и порожденных ими
потоков.

Исполняемый программный код процесса считывается из выполняемого файла (executable) и выполняется в адресном пространстве пользователя. Когда
программа выполняет системный вызов, или возникает исключительная ситуация, программа входит в пространство ядра. С этого момента говорят, что
ядро выполняет программу от имени процесса и делает это в контексте процесса. В контексте процесса макрос current является действительным. Любые обращения к ядру из процесса возможны только через интерфейсы системных вызовов и обработчиков исключительных ситуаций.

Поле state структуры task_struct описывает состояние процесса. Каждый
процесс в системе может находиться в одном из пяти состояний:

* TASK_RUNNING процесс готов к выполнению (runnable). Он либо выполняется в данный момент, либо находится в одной из очередей на выполнение.

* TASK_INTERRUPTIBLE процесс приостановлен (находится в состоянии ожидания, sleeping). Процесс в этом состоянии ожидает выполнения некоторого условия. Когда условие выполнится, ядро переводит процесс в состояние TASK_RUNNING. Процесс может возобновить выполнение также при получении им некоторого сигнала.

* TASK_UNINTERRUPTIBLE аналогично TASK_INTERRUPTIBLE, однако процесс не возобновляет выполнение при получении сигнала. Используется в том случае, если процесс должен работать беспрерывно или когда некоторое событие может возникать достаточно часто. Используется реже , чем TASK_INTERRUPTIBLE.

* TASK_ZOMBIE процесс завершен, однако дескриптор процесса должен оставаться доступным на тот случай, если родительскому процессу потребуется получить доступ к этому дескриптору. Дескриптор освобождается, когда родительский процесс вызывает wait4().

* TASK_STOPPED выполнение процесса приостановлено. Задача не выполняется, и не может выполняться. Такое может случиться, если процесс получает сигнал SIGSTOP, SIGTSTP, SIGTTOU, или если сигнал приходит в тот момент, когда процесс находится в состоянии отладки.

Ядро может менять состояние процесса. Предпочтительно использовать для этого функцию __set_task_state(). Определение функции для платформы i386 находится в include/linux/sched.h:

```
#define __set_task_state(tsk, state_value)          \
    do { (tsk)->state = (state_value); } while (0)
    /*задание tsk установить в состояние state_value*/
```

Создание процесса

В большинстве операционных систем для создания процесса используется метод порождения (spawn). Процесс создается в новом адресном пространстве, в который считывается исполняемый файл, и после этого происходит

исполнение процесса. В Unix (и Linux) эти операции разбиты на две функции:
fork() и exec().

Традиционно функция fork() делала дубликат всех всех родительских ресурсов и передавала их порожденному. Однако этот подход достаточно неэффективный. В Linux вызов fork() реализован с применением технологии
копирования при записи (copy-on-write) страниц памяти. В этом случае родительский и порожденный процессы используют одну копию адресного пространства. Данные помечаются таким образом, что если один из процессов
пытается изменить данные, то создается дубликат, и каждый процес получает
свою копию. До этого они используются read-only.

Реализация системного вызова fork() для платформы i386 находится в файле
arch/i386/kernel/process.c. В Linux он реализован через функцию do_fork()
(функция do_fork() реализована в файле kernel/fork.c).

```
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &#174;s, 0, NULL,
NULL);
}
```

Функция do_fork() свидетельство происхождения Linux от Minix, где этот
вызов также реализован через функцию do_fork(). Функция do_fork() вызывает
функцию copy_process() (реализована в kernel/fork.c) и запускает новый процесс
на выполнение. В функции copy_process() выполняются следующие действия:

* создается стек для нового процесса с помощью dup_task_struct(). Дескрипторы родительского и порожденного процессов на этом этапе идентичны.

* различные поля дескриптора порожденного процесса очищаются или устанавливаются в начальные значения. На этом этапе устанавливается
значение поля tgid.

* в зависимости от переданных флагов, решается какие ресурсы будут общими, а какие уникальными для процессов.

После всех действий функция возвращает указатель на новый процесс. Далее происходит возврат в do_fork(). Если возврат copy_process() происходит
успешно, то новый порожденный процесс продолжает выполнение. В зависимости от флагов и их комбинаций, передаваемых в функцию do_fork,

можно создавать различные типы процессов. Возможные флаги приведены в таблице 2.5.

Таблица 2.5 Флаги функции do_fork()

Флаг	Расшифровка
CLONE_FILES	Родительский и порожденный процессы совместно используют открытые файлы
CLONE_FS	Родительский и порожденный процессы совместно используютинформацию о файловой системе
CLONE_IDLETASK	Установить значение PID в нуль (используется только для холостых (idle) задач)
CLONE_NEWNS	Создать новое пространство имен для порожденной задачи (см. параграф 2.4.1)
CLONE_PARENT	Родительский процесс порождающего процесса становится родительским процессом для порожденного процесса
CLONE_PTRACE	Продолжить трассировку и для порожденного процесса (используется для отладки)
CLONE_SIGHAND	У порожденного и родительского процессов будет общие обработчики сигналов
CLONE_THREAD	Родительский и порожденный процессы будут принадлежать одной группе потоков
CLONE_UNTRACED	Запретить родительскому процессу использование флага CLONE_PTRACE для порожденного процесса
CLONE_STOP	Запустить процесс в состоянии TASK_STOPPED
CLONE_VM	У порожденного и родительского процессов будет общее адресное пространство

Поток в пространстве ядра может быть создан только другим потоком, работающим в пространстве ядра, функцией kernel_thread() (реализована в
arch/i386/process.c):

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags);
```

Новый поток выполняет функцию с адресом fn, и параметром arg. Поток создается с помощью вызова do_fork() с флагами flags:

```
return do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, &#174;s, 0,NULL, NULL);
```

Функция потока в этом случае обычно содержит замкнутый цикл: поток выполняется по мере необходимости, переходит в приостановленное состояние,
и выполняется снова.

Завершение процесса

По завершении процесса ядро должно освободить занимаемые ресурсы. Обычно уничтожение происходит, когда вызывается exit() (компилятор C помещает его вызов в конце функции main()). Однако процесс может завершиться непроизвольно: при получении сигнала, или возникновении ситуации, которую процесс не может обработать. В любом случае, основную

работу по завершению процесса выполняет функция do_exit() (реализована в
kernel/exit.c).

После завершения функции do_exit() процесс не может выполняться. Но
дескриптор все еще занимает память. Функция release_task() окончательно
очищает дескриптор процесса. Если родительский процесс завершается до того,
как завершился один из его потомков, то "беспризорным" процессам
назначается новый родитель (один из родственных потоков) или процесс init.
Процесс init периодически удаляет все назначенные ему дочерние процессы
зомби. [10]

2.3.2. Адресное пространство процесса

Адресное пространство процесса состоит из диапазона адресов, выделенных
процессу и которые он может использовать. Размер адресного пространства
может быть больше доступной физической памяти. С помощью ядра процесс
может динамически добавлять и удалять области памяти своего адресного
пространства. Обращение за пределы адресного пространства процесса
приводят к ошибке "Segmentation Fault" (аналог этой ошибки в Windows
"Access violation at adress[0000:0000]").

Процесс имеет пять различных областей памяти: [10, 16]

* Код. Область, в которой находятся исполняемые инструкции, называемая
  сегментом кода или сегментом текста (text section).

* Инициализированные данные. Статически выделенные и глобальные
  данные, которые инициализированы ненулевыми значениями, находятся
  в сегменте данных (data section).

* Инициализированные нулями данные (другое название
  неинициализированные данные). Глобальные и статически выделенные
  данные, которые по умолчанию инициализированы нулями, хранятся в
  сегменте BSS (bss section).

* Куча (heap). Куча это область, откуда выделяется динамическая память,
  получаемая с помощью функции malloc() и подобными ей. Адресное
  пространство процесса растет, когда из кучи выделяется память.

* Стек. Определение стека дано в параграфе 1.1. Именно использование

стека для параметров функций и возвращаемых ими значений делает удобным написание рекурсивных функций.

Каждое значение адреса в адресном пространстве принадлежит только одной области памяти, и области памяти не перекрываются.

Дескриптор памяти

Ядро представляет адресное пространство в виде структуры mm_struct, описанной в файле include/linux/sched.h, и называемой дескриптором памяти.
Все структуры mm_struct объединены в двухсвязный список с помощью полей
mmlist. Первый элемент этого списка дескриптор init_mm процесса init. Общее количество дескрипторов хранится в глобальной переменной mmlist_nr.
Указатель на дескриптор памяти хранится в поле mm дескриптора структуры
task_struct. Для копирования дескриптора памяти родительского процесса в
дескриптор памяти порожденного процесса используется функция copy_mm().

Если при создании процесса указывается флаг CLONE_VM (создание потока), то новый дескриптор не создается. Вместо этого в поле дескриптора
порожденного процесса записывается указатель на дескриптор родительского
процесса:

```
oldmm = current->mm; /*дескриптор текущего процесса*/
if (clone_flags & CLONE_VM) { /*указан флаг CLONE_VM*/
    atomic_inc(&oldmm->mm_users); /*увеличение на 1*/
    mm = oldmm; /*записывается указатель*/
    goto good_mm;
}
good_mm:
tsk->mm = mm; /*присвоение указателя*/
```

Когда процесс завершается, вызывается функция mmput(), которая уменьшает на единицу значение mm_users дескриптора памяти. Когда значение
mm_users становится равным нулю, вызывается функция mmdrop(), которая уменьшает на единицу значение поля mm_count. Когда значение mm_count становится равным нулю, вызывается функция free_mm(), которая освобождает
дескриптор памяти.

Потоки пространства ядра

Потоки пространства ядра не имеют своего адресного пространства (значение поля mm равно NULL), поскольку они не обращаются к страницам памяти в пространстве пользователя. Чтобы обеспечить потоки ядра необходимыми данными, каждый поток использует дескриптор памяти задания,
которое выполнялось перед ним.

Когда поток ядра планируется на выполнение, ядро определяет, что значение поля равно NULL, и оставляет загруженным предыдущее адресное пространство. Поскольку потоки ядра не обращаются к памяти в пространстве
пользователя, то они используют только ту информацию об адресном пространстве, которая является общей для всех процессов. Например, при необходимости поток ядра может использовать таблицы страниц предыдущего
процесса.

Таблицы страниц

Когда приложение обращается к адресу виртуальной памяти, этот адрес должен быть конвертирован в физический. Соответствующий поиск
выполняется с помощью таблиц страниц: виртуальный адрес разбивается на части; каждая часть используется в качестве индекса (номера) записи в таблице.
Таблица содержит или указатель на другую таблицу, или указатель на соответствующую страницу физической памяти.

В операционной системе Linux таблицы страниц состоят из трех уровней
(даже для тех платформ, которые аппаратно не поддерживают трехуровневых
таблиц):

1. Таблица страниц верхнего уровня называется каталогом страниц (page
   global directory, PGD). Представляет собой массив элементов pgd_t (соответствует типу unsigned long). Записи в таблице PGD содержат указатели на каталоги страниц более низкого уровня, PMD.

2. Каталоги страниц второго (среднего) уровня (page middle direcory) это
   массивы элементов типа pmd_t. Записи таблиц PMD указывают на таблицы PTE.

3. Таблицы страниц последнего уровня называются просто таблицами страниц (page table entry) содержат элементы pte_t. Эти записи указывают
   на страницы физической памяти.

Каждый процесс имеет свои таблицы страниц (потоки использую общую таблицу). Поле pgd дескриптора памяти указывает на глобальный каталог страниц.

Для большинства платформ поиск в таблицах страниц выполняется аппаратно (хотя бы частично). Поскольку каждое обращение к виртуальной памяти требует преобразования соответствующего адреса физической памяти,
операции с таблицами страниц должны выполняться очень быстро. Поэтому большинство процессоров имеют буфер быстрого преобразования адреса (translation lookaside buffer, TLB), работающий как аппаратный кэш
отображения виртуальных адресов на физические. Перед обращением к

виртуальному адресу процессор проверяет это отображение в TLB. В случае
успеха сразу же возвращается физический адрес, в противном случае преобразование адреса происходит с помощью таблиц страниц. [10]

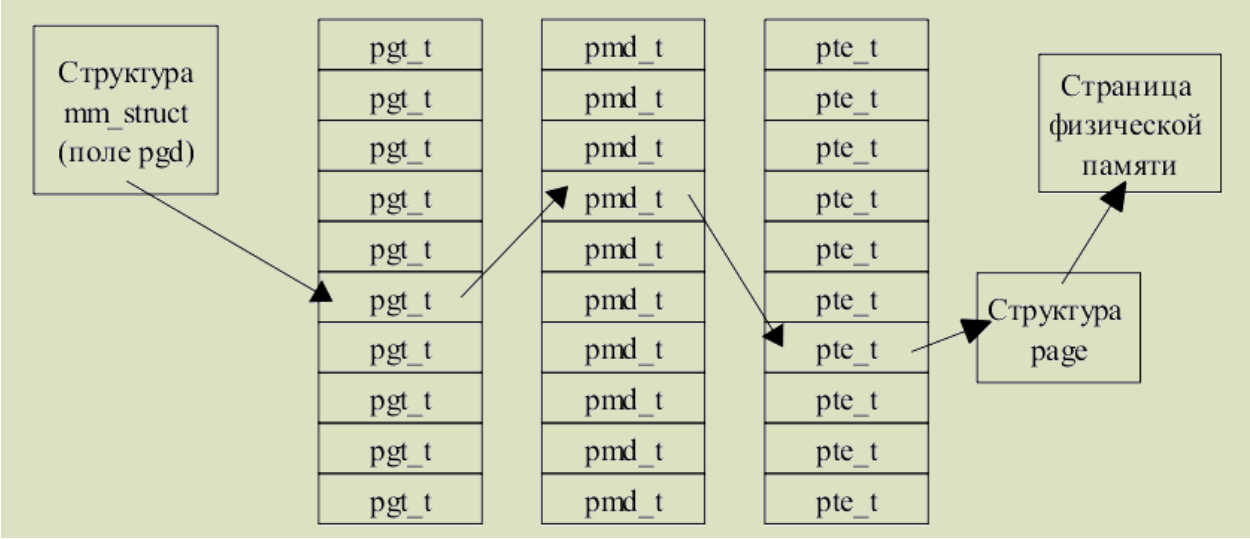

Рис. 2.2. Таблицы страниц.

2.3.3. Работа с дескриптором процесса

Задание

Создать модуль, который будет отображать основную информацию о процессе: PID, UID, адреса сегментов и др. Эту информацию модуль должен
получать непосредственно из дескриптора процесса.

Для модуля предусмотреть возможность в виде параметра передать PID процесса, информация о котором будет отображаться. Если параметр не передается, модуль будет отображать информацию о процессе с PID равным единице (процесс init, который есть в любой Linux системе).

Ход работы

Указатель на занимаемое адресное пространство хранится в поле mm_struct
active_mm дескриптора процесса. Адреса занимаемой области памяти хранятся
в поле mmap структуры mm_struct. Поле mmap имеет тип struct vm_area_struct.
Определение структур task_struct и mm_struct хранится в файле
include/linux/sched.h. Определение структуры vm_area_struct находится в файле
include/linux/mm.h:

```
#include <linux/sched.h>
#include <linux/mm.h>
```

По умолчанию будем выводить информацию о процессе init (этот процесс
есть в любой Linux системе). Если мы захотим обратиться к процессу с иным
значением идентификатора, нам потребуется переменная типа pid_t. Тип pid_t
определяется в файле include/linux/types.h.

```
#include <linux/types.h>
static int nr = 1;
```

Сама переменная объявляется типом int. Это необходимо, если мы хотим
передать ее в качестве параметра модулю. Параметры, передаваемые модулю
могут быть только типов: byte, short, ushort, int, uint, long, ulong, charp (строка),
bool, invbool. Передача параметров модулю производится с помощью макросов
module_param(nr, int, 0000) (имя, тип, права доступа) и
MODULE_PARM_DESC(nr, "Process ID") (имя, описание). Определения
макросов находятся в файле include/linux/moduleparam.h.

```
module_param(nr, int, 0000);
MODULE_PARM_DESC(nr, "Process ID");
```

Если модуль не был собран как часть ядра (забегая вперед, скажем, что
делать этого нельзя), то мы можем передавать параметры. Иначе информация
будет отображаться только о процессе init.

Нам потребуется переменная типа task_struct, хранящая дескриптор процесса, с которым мы работаем. Функция find_task_by_pid() возвращает дескриптор процесса с соответствующим значением pid. Для этой функции мы
и определили тип pid_t.

```
struct task_struct *task;
task = find_task_by_pid((pid_t)nr);
```

Извлекать информацию из дескриптора теперь можно обращением к переменной task. Например, если мы хотим узнать PID или состояние процесса,
мы должны обратиться к полям pid и state соответственно:

```
printk(KERN_INFO "taskinfo: Process %d information:\n",
task->pid);
printk(KERN_INFO "taskinfo: STATE  %u\n", (u32)task-
>state);
```

Значения начальных и конечных адресов занимаемых сегментов хранятся в
поле active_mm дескриптора процесса и имеют тип unsigned long. Это поля
start_code и end_code для сегмента кода, start_data и end_data для сегмента
данных, start_brk и brk для "кучи". Длина сегмента определяется по формуле
(начальный_адрес конечный_адрес). Создадим переменную типа mm_struct
для обращения к этим полям и присвоим ей значение поля active_mm.

```
struct mm_struct *a_mm;
a_mm = task->active_mm;
```

Обращение к адресам сегментов теперь можно производить через
переменную a_mm. Например, чтобы узнать начальный адрес, конечный адрес
и длину сегмента кода:

```
printk(KERN_INFO "taskinfo: START CODE SEGMENT ADDRESS
0x%08X\n", (u32)a_mm->start_code);
printk(KERN_INFO "taskinfo:       END CODE SEGMENT ADDRESS
0x%08X\n", (u32)a_mm->end_code);
printk(KERN_INFO "taskinfo: CODE SEGMENT LENGTH
0x%08X\n",(u32)(a_mm->end_code  a_mm->start_code));
```

Значение занимаемых адресов памяти хранится в поле mmap поля
active_mm дескриптора и имеет тип unsigned long. Это поля vm_start и
vm_end.
Создадим переменную типа struct vm_area_struct для обращения к адресному
пространству процесса. Переменной присвоим значение поля mmap (task->active_mm->mmap).

```
struct vm_area_struct *vma;
vma = task->active_mm->mmap;
```

Теперь обращение к полям vm_start и vm_end можно производить через
переменную vma.

```
printk(KERN_INFO "taskinfo: START MEMORY AREA ADDRESS
0x%08X\n", (u32)vma->vm_start);
printk(KERN_INFO "taskinfo:          END MEMORY AREA
ADDRESS 0x%08X\n", (u32)vma->vm_end);
```

Соответственно, вся область памяти, занимаемая процессом, высчитывается
по формуле (vm_end vm_start):

```
printk(KERN_INFO "taskinfo: MEMORY AREA LENGTH
0x%08X\n",(u32)(vma->vm_end  vma->vm_start));
```

Чтобы отобразить информацию из определенного дескриптора, модулю при загрузке передается значение параметра nr. Чтобы передать модулю значение параметра nr, модуль нужно загружать иначе.

```
insmod taskinfo.ko nr=<pid>
```

Требуемая информация будет отображаться в /var/log/messages. Чтобы выделить из файла только то, что относится к модулю, обращаться к файлу следует так:

```
cat /var/log/messages | grep taskinfo: | tail -n 21
```

Команда отобразит только последнюю 21 строку, содержащую подстроку "taskinfo:". Весь код модуля находится в файле taskinfo.c приложений.

Указание

Данный модуль нельзя собирать вместе с ядром, поскольку при загрузке ядра нет никакого процесса init (он просто не успел появиться). Модуль можно компилировать только как модуль. Подобные модули, которые нельзя компилировать вместе с ядром, отсутствуют в ядре Linux, и здесь он приведен только как пример.

2.4. Виртуальная файловая система (VFS)

2.4.1. Подсистема VFS

Общий интерфейс к файловым системам

Виртуальная файловая система (Virtual File System), иногда называемая виртуальным файловым коммутатором (Virtual File Switch) это подсистема ядра, реализующая интерфейс пользовательских программ к файловой системе.
Все файловые системы зависят от VFS, что позволяет им совместно функционировать.

Подсистема VFS позволяет системным вызовам, таким как open(), read() и write(), работать независимо от файловой системы и физической среды носителя. Общий интерфейс возможен только благодаря тому, что в ядре реализован обобщающий уровень, скрывающий низкоуровневый интерфейс файловых систем: VFS реализует общую файловую модель, представляющую общие функции и особенности работы потенциально возможных файловых систем. Эта файловая модель похожа на файловую систему Unix.

Обобщающий уровень работает путем определения базовых интерфейсов и структур данных, необходимых для поддержки всех файловых систем. Поддержка формирует се концепции работы с файловой системой, такие как "открытие файла", "элемент каталога" и др. Все детали скрываются в коде
самой файловой системы. Таким образом, по отношению к остальным частям ядра все файловые системы поддерживают одинаковые функции.

Например, рассмотрим библиотечный библиотечный вызов write(f, &buf, len) запись len байт по адресу buf в файл, представляемый дескриптором f.
Этот вызов реализован с помощью функции ядра sys_write() (реализована в
fs/read_write.c), которая определяет функцию записи для той файловой системы, на которой находится файл, представленный дескриптором f. Системный вызов sys_write() вызывает функцию записи VFS vfs_write() (реализована в fs/read_write.c), которая вызывает функцию файловой системы
физического носителя, отвечающую за запись (file->f_op->write()). Схема
работы приведена на рисунке 2.3.

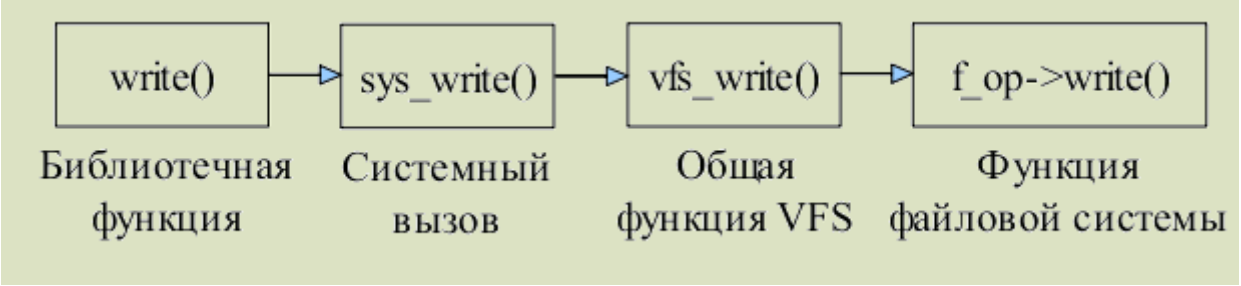

Рис. 2.3. Схема работы функции write().

Объекты VFS

Реализация подсистемы VFS имеет черты объектно ориентированного подхода. Общая модель представлена в виде структур данных, похожих на объекты: структуры данных содержат указатели на элементы данных и на функции, которые работают с этими данными. Существует четыре основных типа объектов VFS:

* Объект суперблок (superblock), представляющий одну смонтированную
файловую систему.

* Объект файловый индекс (inode), представляющий определенный файл.

* Объект элемент каталога (dentry), представляющий определенный
элемент каталога. При этом не существует специальных объектов для
каталогов. Объект dentry представляет собой фрагмент пути, который
может содержать файл. Но dentry это не каталог, и каталог это не

dentry.

* Объект файл (file), представляющий открытый файл, связанный с процессом.

Каждый из объектов VFS содержит объект операций, которые описывающие методы, которые ядро может применять для основных объектов:

* Объект super_operations (операции с суперблоком ФС). Структура описана в файле include/linux/fs.h. Содержит методы, которые ядро может вызывать для определенной файловой системы.

* Объект inode_operations (операции с файловыми индексами). Структура описана в файле include/linux/fs.h. Содержит методы, которые ядро вызывать для определенного файла.

* Объект dentry_operations (операции с элементами каталогов). Структура описана в файле include/linux/dcache.h. Содержит методы, которые ядро может применять для определенного элемента каталогов.

* Объект file_operations (операции с файлами). Структура описана в файле include/linux/fs.h. Содержит операции, которые ядро может вызывать для открытого файла.

Объекты операций реализованы в виде структур, содержащих указатели на функции, оперирующих объектом VFS. Если достаточно базовой функциональности, то объект может унаследовать общую функцию, в противном случае файловая система присваивает указателям адреса своих собственных методов.

Объект superblock

Объект суперблок описывает определенную файловую систему и реализуется для каждой файловой системы. Обычно соответствует физическому суперблоку или создается "на лету" (например, для файловой системы procfs) и хранится в памяти.

Суперблок представляется с помощью стуктуры super_block, определенной

в файле include/linux/fs.h. Объект суперблок создается и инициализируется
функцией alloc_super() при монтировании файловой системы.

Поле s_op это указатель на таблицу операций суперблока, представленной
структурой super_operations. Каждое поле структуры s_op это указатель на
функцию, работающую с объектом суперблок. Если, например, файловой системе необходимо выполнить запись в суперблок, то вызывается следующая
функция.

```
sb->s_op->write_super(sb);
```

Параметр sb указатель на суперблок. Функции write_super() передается
указатель на суперблок в качестве параметра, несмотря на то, что метод связан
с суперблоком. Это очень похоже на объектно ориентированный подход. На
языке C++ это выглядело бы так:

```
sb.write_super();
```

Но ядро Linux создается при помощи языка C, поэтому разработчики не могут использовать явно средства объектно ориентированного
программирования.

Объект inode

Объект inode содержит информацию, которая необходима ядру для манипуляций с файлами и каталогами (каталог в Unix это по сути файл). Для
файловых систем Unix объект inode подсистемы VFS считывается из дисковых
индексов. Если файловая система не имеет индексов (FAT, HPFS), то эта информация получается из других дисковых структур.

Объект inode представлен структурой inode, определенной в файле include/linux/fs.h. Для каждого файла в системе существует его индекс, однако
объект файлового индекса в памяти создается только тогда, когда к этому
файлу осуществляется доступ. Если файловая система не поддерживает некоторые свойства, то его можно реализовать как угодно или сделать нулевым
(NULL). Поле i_op указывает на операции с файловым индексом.

Объект dentry

Рассмотрим путь к файлу в системе Unix. Например, путь к утилите ls выглядит так: /bin/ls. В этом пути bin, ls это файлы, но bin это специальный

файл, который является каталогом. Для представления обоих этих элементов
служат объекты файловых индексов. Однако подсистеме VFS необходимо выполнять операции, специфичные для каталогов: поиск элемента, проверка
существования, переход на следующий компонент и др.

Для этого в подсистеме VFS реализован элемент элемента каталога (directory entry или dentry). Объект dentry это определенный элемент пути. /,
bin, ls это элементы каталога. При выполнении операций с каталогами подсистема VFS формирует объекты dentry "на лету".

Объект dentry представляется структурой dentry, определенной в файле
include/linux/dcache.h. Указатель на структуру операций с объектом dentry
хранится в поле d_op. Объект dentry не соответствует никакой реальной структуре данных на физическом носителе.

После нахождения пути к объекту, система может сохранить найденные объект dentry в кэше для облегчения поиска впоследствии. Кэш объектов dentry
называется dcache (Dentry Cache) и состоит из трех частей:

* Список используемых объектов dentry, связанных с определенным файловым индексом (поле i_dentry структуры inode).

* Двухвязный список неиспользуемых и негативных объектов dentry. Список отсортирован по времени: в начале списка находятся самые новые элементы. Если ядро удаляет элементы из этого списка, то они
берутся из конца (самые старые элементы).

* Хэш-таблица (массив dentry_hashtable) и хэш-функция (поле d_hash
структуры dentry_operations), которые позволяют быстро преобразовать
заданный путь в объект dentry.

Объект элемента каталога может быть в одном из трех состояний:

* Используемом (used). Поле d_inode указывает на объект inode, и подсистема VFS использует данный элемент каталога (поле d_count больше 0). Такие объекты не могут быть удалены.

* Неиспользуемом (unused). Поле d_inode указывает на объект inode, но
подсистема VFS не использует его в данный момент (поле d_count равно
0). Данный объект может быть удален, но может и храниться в памяти "на
всякий случай".

* Негативном (negative). Поле d_inode не указывает на объект inode (файл
удален, или такого пути не существует). Такие объекты сохраняются для
поиска в будущем, но могут быть удалены.

Объект file

Объект файл это представление открытого файла, хранящееся в оперативной памяти. Объект создается в системном вызове open() и уничтожается в системном вызове close(). Для одного файла может существовать несколько объектов file, так как одновременно к одному файлу
может обращаться несколько процессов. Файловый объект представляется структурой file, определенной в include/linux/fs.h. Объект file не соответствует
никакой реальной структуре на физическом носителе. Он содержит указатель
на связанный с ним объект dentry, который содержит указатель на связанный с
ним объект inode, представляющей физический файл. Указатель на операции с
объектом file хранится в поле f_op.

Другие объекты VFS

Каждая зарегистрированная файловая система представлена структурой file_system_type, определенной в include/linux/fs.h. Структура описывает
файловую систему и ее свойства.

Каждая точка монтирования представлена структурой vfsmount. Структура
используется для представления конкретной файловой системы, или точки монтирования, и определена в include/linux/mount.h. Структура содержит информацию о точке монтирования, включая положение и флаги, с которыми эта точка была монтирована.

Каждый процесс имеет в дескрипторе три структуры, которые описывают
файловую структуру и файлы, связанные с процессом: fs_struct, files_struct и
namespace.

Структура fs_struct определена в include/linux/fs_struct.h. Структура
содержит текущий рабочий каталог и корневой каталог процесса.

Структура files_struct определена в include/linux/file.h. В структуре хранится
информация об открытых файлах и файловых дескрипторах. Список открытых файловых объектов хранится в массиве fd структуры fdtable (определена в
include/linux/file.h).

Структура namespace содержит пространства имен, индивидуальные для

процесса (то есть процесс может иметь уникальную иерархию смонтированных
файловых систем). Структура определена в файле include/linux/namespace.h.
Список смонтированных файловых систем, которые составляют пространство имен, хранится в поле list.

Процессы, созданные с помощью флагов CLONE_FILES и CLONE_FS (например, потоки), имеют общие с родительским структуры files_struct и
fs_struct. Все процессы по умолчанию используют одинаковое пространство
имен. Если при создании процесса указан флаг CLONE_NEWNS, для процесса создается уникальная копия пространства имен. Но обычно этот флаг не указывается. [10]

2.4.2. Создание файловой системы

Задание

Создать модуль, реализующий новую файловую систему. Файловая система
должна работать с файлами, хранящими количество обращений к ним. После создания файловой системы смонтировать ее в каталог /linfs/.

Ход работы

Для того, чтобы создать файловую систему, мы должны определить необходимые объекты VFS и операции с ними. Назовем новую файловую систему LinFS. Общий порядок действий таков:

1. Для работы с файлами и каталогами необходимо определить функцию создания объекта inode.
2. Чтобы ядро смогло выполнять все определенные в файловой системе операции, файловую систему надо зарегистрировать в ядре.
3. Для файловой системы необходимо реализовать функции создания файлов и каталогов.
4. После этого реализуются функции операций с файлами и суперблоком.
5. После этого файловую систему можно монтировать.

Создание объекта inode

Прежде всего необходимо реализовать функцию инициализации inode для файловой системы. Это функция linfs_make_inode().

```
static struct inode *linfs_make_inode(struct super_block *sb, int mode)
```

Функция принимает указатель на суперблок файловой системы и значение

прав доступа для нового inode и возвращает указатель на созданный inode.
Память для нового inode выделяется функцией функцией new_inode().

```
struct inode *ret = new_inode(sb);
```

В случае успешной инициализации (if (ret)) заполняются поля структуры
inode:

```
ret->i_mode = mode;
ret->i_uid = ret->i_gid = 0;
ret->i_blksize = PAGE_CACHE_SIZE;
ret->i_blocks = 0;
ret->i_atime = ret->i_mtime = ret->i_ctime = CURRENT_TIME;
```

Поле i_mode права доступа к объекту inode. Устанавливаемое значение
поля передается во втором параметре функции. Поле i_uid и i_gid User ID и
Group ID нового объекта inode соответственно. Обычно для этих полей
устанавливается значение нуля. Поле i_blksize размер блока (кластера) в
байтах. Его размер устанавливается в значение PAGE_CACHE_SIZE,
определенное в файле include/linux/pagemap.h через постоянную PAGE_SIZE,
определенную в файле include/asm/page.h и равную 4096 (байт). Поле i_blocks
размер файла в блоках. Поскольку пока файл не создается, значение поля
равняется нулю. Поля i_atime, i_mtime, i_ctime соответственно время
последнего доступа к файлу, время последнего изменения в файле и время
изменения индекса. Значение этих полей устанавливается в текущее значение
времени, поскольку индекс только что был создан.

Регистрация файловой системы

При загрузке модуля новая файловая система регистрируется в ядре. Это
делается вызовом функции register_filesystem(). В качестве параметра функции
передается структура file_system_type.

```
static struct file_system_type linfs_type = {
     .owner     = THIS_MODULE,
     .name      = "linfs",
     .get_sb    = linfs_get_super,
     .kill_sb   = kill_litter_super,
};
```

Структура описывается синтаксисом ISO C99. Предпочтительнее
использовать этот синтаксис. Другой способ определения структуры:

```
static struct super_operations linfs_sops = {
     owner:     THIS_MODULE,
     name:      "linfs",
     get_sb:    linfs_get_super,
     kill_sb:   kill_litter_super,
};
```

Мы будем придерживаться синтаксиса ISO C99.

Поле name имя файловой системы: ext2, vfat, ntfs все это имена, определенные в поле name других файловых систем. Поле get_sb функция, которая используется для считывания суперблока с диска. Функция
linfs_get_super() реализована через стандартную функцию
get_sb_single().

```
static struct super_block *linfs_get_super
     (struct file_system_type *fst, int flags, char
*devname, void *data, struct vfsmount *mnt)
 {
          return get_sb_single(fst, flags, data,
linfs_fill_super, mnt);
 }
```

Функция get_sb_single() реализована в файле fs/super.c.

```
int get_sb_single(struct file_system_type *fs_type, int
flags, void *data,
     int (*fill_super)(struct super_block *, void *, int),
struct vfsmount *mnt)
```

Функция возвращает указатель на соответствующий объект суперблока файловой системы fs_type. При этом выполняется функция, на которую указывает параметр fill_super(). В нашем случае он указывает на функцию
linfs_fill_super().

```
static int linfs_fill_super(struct super_block *sb, void
*data, int silent)
```

В функции сначала заполняются поля объекта суперблока.

```
sb->s_blocksize = PAGE_CACHE_SIZE;
sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
sb->s_magic = LFS_MAGIC;
sb->s_op = &linfs_sops;
```

Поля s_blocksize и s_blocksize_bits размер блока (кластера) в байтах и
битах соответственно. Поле s_op ссылка на объект операций с суперблоком
(рассматривается ниже). Поле s_magic "магический номер" файловой

системы. Он определяется как 0x19980122.

```
#define LFS_MAGIC 0x19980122
```

Далее в функции создается корневой каталог.

```
struct inode *root;
struct dentry *root_dentry;
root = linfs_make_inode (sb, S_IFDIR | 0755);
root->i_op = &simple_dir_inode_operations;
root->i_fop = &simple_dir_operations;
```

Для представления корневого каталога необходимо создать для него новый
объект inode с помощью функции linfs_make_inode(). Поля i_op и i_fop указатели на объект операций с объектом inode и файловых операций соответственно. Значения simple_dir_inode_operations и simple_dir_operations
определены в файле fs/libfs.c.

Чтобы объект dentry "отображался в ядре", его необходимо "увязать" с
супреблоком.

```
root_dentry = d_alloc_root(root);
sb->s_root = root_dentry;
linfs_create_files(sb, root_dentry);
```

Поле s_root указатель на объект dentry, представляющий корневой каталог. После создания корневого каталога в файловой системе создаются
первые файлы вызывается функция
linfs_create_files(). В качестве
параметров функция принимает указатель на суперблок файловой системы и корневой объект dentry.

Создание файлов

В функции linfs_create_files() сначала создается файл counter.

```
atomic_set(&counter, 0);
linfs_create_file(sb, root, "counter", &counter);
```

Функция atomic_set() выполняет атомарную
(неделимую) операцию
присвоения переменной counter типа atomic_t значения 0. Функция
linfs_create_file() принимает в качестве параметров объект суперблока, создаваемый объект dentry (каталог), в котором создается файл, имя создаваемого объекта dentry и текущее значение счетчика.

```
static struct dentry *linfs_create_file
```

```
            (struct super_block *sb, struct dentry *dir, const char
*name, atomic_t *counter)
```

В функции linfs_create_file() сначала заполняется кэш объектов dentry.

```
        struct qstr qname;
        qname.name = name;
        qname.len = strlen(name);
        qname.hash = full_name_hash(name, qname.len);
```

Структура qstr элемент кэша объектов dentry, определенная в файле include/linux/fs.h. Структура содержит имя объекта и дополнительную информацию об объекте: поле len (здесь это длина имени) и хэш-значение.
Функция full_name_hash() реализована в файле include/linux/dcache.h.

Далее создаются объекты inode и dentry и связываются друг с другом.

```
        dentry = d_alloc(dir, &qname);
        inode = linfs_make_inode(sb, S_IFREG | 0644);
        inode->i_fop = &linfs_fops;
        inode->u.generic_ip = counter;
        d_add(dentry, inode);
```

Функция d_alloc() выделяет память для нового объекта dentry. Функция
d_alloc() реализована в файле fs/dcache.c. В качестве параметров функция
принимает указатель на создаваемый объект dentry. Затем создается новый
объект inode. Поле u.generic_ip это специфическая для файловой системы
информация. Поле устанавливается в значение счетчика. После этого в функции d_add() происходит связка этих объектов. Функция принимает в качестве параметра указатель на объект dentry и объект inode. Объект dentry
добавляется в хэш-таблицу, объект inode "присоединяется" (attach) к объекту
dentry. Функция реализована в файле include/linux/dentry.h.

После создания файла counter в функции linfs_create_files() создается
подкаталог subdir с файлом subcounter.

```
        atomic_set(&subcounter, 0);
        subdir = linfs_create_dir(sb, root, "subdir");
```

Функция linfs_create_dir() создает каталог в файловой системе. Функция
принимает в качестве параметров указатель на объект суперблок, указатель на
объект dentry, в котором создается каталог и имя каталога.

Функция аналогична функции linfs_create_file(), поскольку каталоги в Unix
по своей сути тоже файлы. Единственное отличие в заполнении полей объекта inode.

```
inode->i_op = &simple_dir_inode_operations;
inode->i_fop = &simple_dir_operations;
```

Поля i_op и i_fop устанавливаются в значение simple_dir_inode_operations и
simple_dir_operations. Если подкаталог успешно создан (if (subdir)), то в нем
создается файл subcounter.

```
linfs_create_file(sb, subdir, "subcounter", &subcounter);
```

Мы рассмотрели функции для создания файлов. Теперь рассмотрим реализацию операций с суперблоком.

Операции с суперблоком

Для реализации операций с суперблоком необходимо создать объект структуры super_operations и "перекрыть" необходимые функции.

```
static struct super_operations linfs_sops = {
     .statfs            = simple_statfs,
     .drop_inode        = generic_delete_inode,
};
```

Поле statfs функция, которая вызывается VFS для получения статистики
файловой системы. Поле хранит ссылку на функцию simple_statfs(),
реализованную в файле fs/libfs.c. Поле drop_inode функция, которая вызывается подсистемой VFS, когда исчезает последняя ссылка на объект inode. Поле хранит ссылку на функцию generic_delete_inode(), реализованную в
файле fs/inode.c.

Операции с файлами

Для определения операций с файлами создадим экземпляр объекта file_operations. Для объекта file_operations определим операции открытия (поле
open), чтения (поле read) и записи (поле write).

```
static struct file_operations linfs_fops = {
     .open= linfs_open,
     .read = linfs_read_file,
     .write = linfs_write_file,
};
```

Соответствующие функции реализации описываются до создания ссылок на
них в объекте. Функция linfs_open() реализуется так.

```
static int linfs_open(struct inode *inode, struct file
*filp)
{
    filp->private_data = inode->u.generic_ip;
    return 0;
}
```

Функция в качестве параметра получает указатель на объект inode, связанный с объектом file с именем filp. Поле private_data структуры file
содержит привязку для драйвера терминала. В функции этот указатель устанавливается в значение поля u.generic_ip, которое было инициализировано
в функции linfs_create_file(). На терминал будет возвращаться значение счетчика, но для этого необходимо реализовать функцию чтения из файла. Функция linfs_read_file() реализует операцию чтения из файла.

```
static ssize_t linfs_read_file
    (struct file *filp, char *buf, size_t count, loff_t
*offset)
```

Функция принимает в качестве параметров указатель на объект file, указатель на буфер, в который передаются данные buf, количество считываемых байт count (размер буфера) и смещение относительно начала файла offset.

Сначала текущее значение счетчика counter считывается и сохраняется в
переменную v с помощью функции atomic_read().

```
Int v;
v = atomic_read(counter);
```

Затем проверяется, что считывание происходит с начала файла (смещение
offset равно нулю). В этом случае счетчик обращений counter увеличивается на
единицу функцией atomic_inc(). В противном случае значение переменной v
уменьшается на единицу.

```
if (*offset == 0)
   atomic_inc(counter);
else
   v -= 1;
```

Значение счетчика записывается в строку tmp длиной до TMPSIZE символов. Длина получившейся строки сохраняется в переменную len.

```
#define TMPSIZE 20
```

```
char tmp[TMPSIZE];
int len;
int len = snprintf(tmp, TMPSIZE, "%d\n", v);
```

Если значения смещения больше длины строки необходимо выйти из функции с кодом возврата 0.

```
if (*offset > len)
   return 0;
```

Если количество запрошенных байт больше, чем можно передать (count > len *offset), значение count устанавливается в точно меньшее значение, равное (len *offset). Отсюда понятен смысл условия (*offset > len).

```
if (count > len  *offset)
   count = len  *offset;
```

Теперь можно передать в пространство пользователя значение переменной tmp, в которую было сохранено значение счетчика. Переменная записывается в область памяти в пространстве пользователя, соответствующее началу смещения.

```
copy_to_user(buf, tmp + *offset, count)
```

Наконец, значение смещения увеличивается на размер буфера count.

```
*offset += count;
```

Функция linfs_write_file() реализует операцию записи в файл.

```
static ssize_t linfs_write_file
  (struct file *filp, const char *buf, size_t count, loff_t
*offset)
```

Функции в качестве параметров передаются указатель filp на структуру file, указатель на записываемую строку buf, размер буфера count и смещение offset.

Реализация функции последовательна. Сначала проверяется, осуществляется ли запись в начало. Если нет (ошибка), выходим с возвращением значения ошибки.

```
if (*offset != 0)
  return -EINVAL;
```

Если количество записываемых байт больше возможного значения TMPSIZE, выходим с возвращением значения ошибки. Если ошибки нет, выделяется память для записываемого значения с помощью функции memset().

```
if (count >= TMPSIZE)
  return -EINVAL;
memset(tmp, 0, TMPSIZE);
```

Считывается значение из пространства пользователя. В случае неудачи выходим с возвращением значения ошибки.

```
if (copy_from_user(tmp, buf, count))
  return -EFAULT;
```

Если на предыдущих этапах не было ошибок, сохраним полученное значение в переменную counter.

```
atomic_set(counter, simple_strtol(tmp, NULL, 10));
```

Функция simple_stroll() реализована в файле lib/sprintf.c. В данном случае
функция конвертирует значение строки tmp в тип unsigned long long с количеством символов 10. В качестве признака окончания строки используется
символ NULL.

В функции выгрузки модуля нужно отменить регистрацию файловой системы с помощью функции unregister_filesystem().

```
unregister_filesystem(&linfs_type);
```

Код модуля находится в файле linfs.c приложений.

Монтирование

Файловая система создана. Как ее монтировать после загрузки модуля и
регистрации в ядре? Файловая система не связана с каким-либо физическим
устройством, поэтому в качестве параметра команде mount вместо названия
устройства носителя передается параметр none.

```
# mount -t linfs none /linfs/
```

Указание

Ошибка в коде может вызвать повреждение или потерю данных на жестком
диске. Все эксперименты настоятельно рекомендуется проводить на виртуальной машине.

2.5. Файловая система procfs

2.5.1. Каталог /proc/

Специальная файловая система procfs существует во всех Unix-системах и
представляет собой интерфейс к структурам данных ядра, доступных пользовательским программам. Файловая система procfs монтируется в каталог
/proc/. Когда происходит обращение к какому-либо файлу в каталоге /proc/,
ядру передается соответствующее сообщение и оно формирует ответ. Таким образом создается иллюзия работы с реальной файловой системой. Ядро и модули ядра могут использовать /proc/, чтобы взаимодействовать с
пользовательскими программами.

Файловая система procfs не связана с физическим носителем и существует
полностью в памяти ядра. Файлам в каталоге /proc/ не соответствуют реальные
объекты на физическом устройстве. Они появляются и исчезают динамически в
процессе работы системы (поэтому их размер равен 0).

Прежде всего, procfs создавалась для отображения информации о процессах
(отсюда и название). Подкаталоги, имена которых состоят только из цифр,
соответствуют процессу, идентификатор pid которого служит именем каталога.
Подкаталог /proc/self/ указывает на текущий процесс. [4, 14]

Каталог /proc/sys/

Каталог /proc/sys/ впервые появился в ядре версии 1.3.57 и применяется в
качестве интерфейса, позволяющего получить доступ к переменным ядра.
Каталог содержит несколько подкаталогов:

- debug/. Содержит отладочную информацию.
- dev/. Содержит информацию, специфичную для устройств (например dev/cdrom/info). В некоторых системах он может быть пуст.
- fs/. Содержит информацию о файловых системах.
- kernel/. Содержит доступ к структурам ядра. В этом каталоге хранятся,
например, тип ОС, версия ядра, имя хоста машины и другое.
- net/. Содержит некоторую информацию по функционированию сетевой подсистемы.
- sunrpc/. Каталог поддерживает удалённый вызов процедур Sun для сетевой файловой системы (NFS). В некоторых системах его нет.

vm/. Каталог содержит файлы для тонкой настройки управления памятью,
буферами и кэшем.

Более подробную информацию о каталоге /proc/ и его подкаталогах можно
получить из справки man proc. [5]

2.5.2. Работа с procfs

Задание

Написать модуль, создающий в каталоге /proc/ новый подкаталог mykernel_test/ с несколькими файлами:

jiffies возвращает количество прерываний таймера со времени загрузки модуля.

jiffies2 символическая ссылка на файл jiffies.

seconds возвращает количество секунд, прошедших со времени загрузки модуля.

foo возвращает последнюю записанную в него строку.

success возвращает надпись "Congratulations! Your Linux experiment successed".

Ход работы

Ядро предоставляет специальные интерфейсы для работы с файловой системой procfs. Объявление функций находится в файле include/linux/proc_fs.h.
Реализация интерфейсных функций для работы с файловой системой procfs находится в каталоге fs/proc/.

Перед тем, как создавать файлы в procfs, необходимо их объявить:

```
static struct proc_dir_entry
    *example_dir,    *jiffies_file,   *symlink,
*seconds_file,   *foo_file, *success_file;
```

Создание каталога происходит с помощью функции proc_mkdir().
Например, создадим каталог /proc/mykernel_test:

```
#define MYKERNEL_ENTRY "mykernel_test"
example_dir = proc_mkdir(MYKERNEL_ENTRY, NULL);
```

Файл, из которого может считывать пользовательский процесс, создается с
помощью функции create_proc_read_entry(). Например, создадим файл jiffies в

каталоге /proc/mykernel_test/ с правами 0444:

```
jiffies_file =
  create_proc_read_entry("jiffies", 0444, example_dir,
proc_read_jiffies, NULL);
```

При попытке чтения вызывается функция proc_read_jiffies():

```
static int proc_read_jiffies
    (char *page, char **start, off_t off, int count, int
*eof, void *data)
{
    int len;
    len = sprintf(page, "Your processor timer ticked %lu
times\n", jiffies);
    return len;
}
```

Функция вернет на терминал строку, на которую указывает переменная page. В данном случае будет выведено сообщение о количестве прерываний системного таймера (это значение хранится в переменной jiffies). Функция
proc_read_seconds() реализована аналогично, но возвращает на терминал значение jiffies/HZ. Постоянная HZ это количество прерываний таймера в
течение секунды. Таким образом jiffies / HZ это количество секунд, которое
работает ядро.

Создать символическую ссылку можно с помощью функции proc_symlink(). Например, создадим символическую ссылку с именем
/proc/mykernel_test/jiffies2 на файл /proc/test/jiffies:

```
symlink = proc_symlink("jiffies2", example_dir, "jiffies");
```

При попытке чтения из этого файла будет вызываться функция, определенная для файла, на который указывает ссылка.

Создать файл, доступный на чтение и запись, можно с помощью функции create_proc_entry(). Например, создадим файл /proc/mykernel_test/foo с правами
доступа 0644:

```
foo_file = create_proc_entry("foo", 0644, example_dir);
```

В этом случае необходимо назначить файлу функции, которые будут вызываться при попытке чтения и записи. Для операций чтения и записи потребуется структура для хранения данных, которая будет связана с полем
данных файла foo.

```
#define FOO_LEN 16
```

```
struct fb_data_t {
     char value[FOO_LEN + 1];
};
strcpy(foo_data.value, "foo\n");
foo_file->data = &foo_data;
```

При попытке чтения будет вызываться функция proc_read_foo(), а при попытке записи будет вызывается функция proc_write_foo():

```
foo_file->read_proc = proc_read_foo;
foo_file->write_proc = proc_write_foo;
```

Функция proc_read_foo() аналогина функции proc_read_jiffies(), но в качестве строки, передаваемой char *page, используется поле value структуры
foo_data_t.

```
static int proc_read_foo(char *page, char **start, off_t
off, int count, int *eof, void *data)
{
    int len;
    struct fb_data_t *fb_data = (struct fb_data_t *)data;
    len = sprintf(page, "Last message: %s", fb_data-
>value);
    return len;
}
```

В функции proc_write_foo() данные для записи передаются из пространства
пользователя с помощью функции copy_from_user().

```
static int proc_write_foo (struct file *file, const char
*buffer, unsigned long count, void *data)
{
    int len;
    struct fb_data_t *fb_data = (struct fb_data_t *)data;
    if(count > FOO_LEN)
           len = FOO_LEN;
    else
           len = count;
    if(copy_from_user(fb_data->value, buffer, len))
           return -EFAULT;
    fb_data->value[len] = '\0';
    return len;
}
```

Для всех файлов необходимо определить поле owner:

```
example_dir->owner = THIS_MODULE;
jiffies_file->owner = THIS_MODULE;
symlink->owner = THIS_MODULE;
foo_file->owner = THIS_MODULE;
```

После сборки и загрузки модуля при обращении к файлу /proc/test/jiffies или
/proc/mykernel_test/jiffies2 на терминал будет выведено сообщение о количестве
срабатываний системного таймера:

```
# cat /proc/mykernel_test/jiffies

Your processor timer ticked 35938 times
```

При обращении к файлу /proc/test/foo будет выведено последнее записанное
сообщение или "foo", если ничего записано не было:

```
# cat /proc/mykernel_test/foo
Last message: foo

# echo "hello" > /proc/mykernel_test/foo

# cat /proc/mykernel_test/foo
Last message: hello
```

При удалении модуля необходимо удалить все созданные структуры с помощью функции remove_proc_entry().

```
remove_proc_entry("success", example_dir);
remove_proc_entry("foo", example_dir);
remove_proc_entry("seconds", example_dir);
remove_proc_entry("jiffies2", example_dir);
remove_proc_entry("jiffies", example_dir);
remove_proc_entry(MYKERNEL_ENTRY, NULL);
```

Полный код модуля находится в файле procfiles.c приложений.