Содержание

2
Начальные сведения о ядре Linux

В этой главе будут рассмотрены основные вопросы, связанные с ядром Linux: где получить исходный код, как его компилировать и как инсталлировать новое ядро. После этого мы рассмотрим отличия между ядром и пользовательскими программам, а также общие программные конструкции, которые используются в ядре. Хотя ядро Linux, вне всякого сомнения, является уникальным во многих отношениях, в конечном итоге оно мало чем отличается от любого другого большого современного программного проекта.

Где взять исходный код ядра

Исходный программный код последней версии ядра Linux всегда доступен как в виде полного архива в формате `.tar` (т.е. в виде архивного файла, созданного утилитой `tar`), так и в виде инкрементальной заплаты по адресу `http://www.kernel.org`.

Если нет необходимости по той или иной причине работать со старыми версиями ядра, то всегда нужно использовать самую последнюю версию. Хранилище `kernel.org` — это то место, где можно найти как само ядро, так и заплаты к нему от ведущих разработчиков.

Использование Git

Несколько лет назад ведущие разработчики ядра Linux под руководством Линуса Торвальдса начали использовать новую систему контроля версий для управления исходным кодом ядра. При создании этой системы, которую Линус назвал *Git*, во главу угла ставилась скорость работы. В отличие от традиционных систем контроля версий, таких как *CVS*, Git является распределенной системой. По этой причине рабочий процесс на ее основе и алгоритм использования оказались совершенно незнакомы большинству разработчиков. Тем не менее я настоятельно рекомендую использовать Git для загрузки исходного кода ядра Linux и управления им.

Чтобы получить с помощью Git копию последней зафиксированной версии дерева ядра Linux, введите приведенную ниже команду.

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

После загрузки исходного кода нужно обновить дерево до состояния последнего обновления, которое сделал Линус. Для этого введите следующее:

```
$ git pull
```

С помощью этих двух команд вы сможете развернуть у себя официальное дерево разработки ядра Linux и всегда поддерживать его в актуальном состоянии. О том, как вносить в него собственные изменения и фиксировать их в хранилище, речь пойдет в главе 20, "Заплаты, хакерство и сообщество". Полное описание возможностей Git выходит за рамки этой книги, однако в Интернете вы найдете большое количество великолепных руководств и инструкций по использованию.

Инсталляция исходного кода ядра

Архив `.tar` исходного кода ядра распространяется в сжатых форматах GNU zip (gzip) и bzip2. Формат bzip2 предпочтительнее, поскольку он обеспечивает больший коэффициент сжатия по сравнению с форматом `gzip`. Архив ядра в формате `bzip2` имеет имя `linux-`*`x.y.z`*`.tar.bz2`, где *`x.y.z`* — номер соответствующей версии исходного кода ядра. После загрузки файла его нужно распаковать с помощью простой команды. Если `.tar`-архив сжат с помощью утилиты `bzip2`, введите такую команду:

```
$ tar xvjf linux-x.y.z.tar.bz2
```

Если сжатие выполнено с помощью `gzip`, то команда должна иметь следующий вид:

```
$ tar xvzf linux-x.y.z.tar.gz
```

Обе эти команды позволяют разархивировать и развернуть дерево исходных кодов ядра в каталог `linux-`*`x.y.z`*. При использовании Git для получения исходного кода ядра и управления им вам не нужно загружать архив в формате `.tar`. Просто введите команду `git clone`, как было описано в предыдущем разделе, и она сама загрузит и распакует самую последнюю версию дерева исходного кода ядра.

> **Где лучше всего инсталлировать и изменять исходный код**
>
> Исходный код ядра обычно инсталлируется в каталог `/usr/src/linux`. Заметим, что это дерево исходного кода нельзя использовать для собственных разработок. Версия ядра, с которой была скомпилирована ваша *библиотека C*, обычно привязана к этому дереву каталогов. Кроме того, чтобы вносить изменения в ядро, не обязательно иметь права пользователя `root`, вместо этого лучше работать в собственном рабочем каталоге и использовать права пользователя `root` только для инсталляции ядра. Даже при инсталляции нового ядра каталог `/usr/src/linux` должен оставаться без изменений.

Использование заплат

В сообществе разработчиков ядра Linux *заплаты* (patch) — это основной *язык общения*. Дело в том, что разработчики обмениваются друг с другом внесенными ими изменениями в исходном коде ядра в виде заплат. При этом наиболее важными являются *инкрементальные заплаты* (incremental patch), которые позволяют перейти от одного дерева ядра к другому. Вместо того чтобы загружать большой архив ядра, можно просто применить инкрементальную заплату и перейти от имеющейся версии дерева ядра к следующей. Это позволяет сэкономить время и не нагружать лишний раз каналы связи. Для

того чтобы применить инкрементную заплату, перейдите к дереву каталогов исходных кодов ядра и введите приведенную ниже команду.

```
$ patch -p1 < ../patch-x.y.z
```

Как правило, заплата, предназначенная для перехода на некоторую версию ядра, должна применяться к предыдущей версии ядра. Процесс генерации и применения заплат будет подробнее описан в следующих главах.

Дерево каталогов исходных кодов ядра

Дерево исходных кодов ядра содержит ряд каталогов, большинство из которых также содержит подкаталоги. Каталоги, которые находятся в корне дерева исходных кодов, и их описание приведены в табл. 2.1.

Таблица 2.1. Дерево исходных кодов ядра

Каталог	Описание
arch	Исходный код, специфичный для аппаратной платформы
block	Слой блочных операций ввода-вывода
crypto	Криптографический API
Documentation	Документация исходного кода ядра
drivers	Драйверы устройств
firmware	Микропрограммы устройств, необходимые для использования некоторых драйверов
fs	Подсистема VFS и отдельные файловые системы
include	Заголовочные файлы ядра
init	Коды для загрузки и инициализации ядра
ipc	Код межпроцессного взаимодействия
kernel	Основные подсистемы ядра, такие как планировщик
lib	Вспомогательные подпрограммы
mm	Подсистема управления памятью и поддержка виртуальной памяти
net	Сетевая подсистема
samples	Примеры и демонстрационный код
scripts	Сценарии компиляции и построения ядра
security	Модуль безопасности Linux
sound	Звуковая подсистема
tools	Вспомогательные утилиты для разработки Linux
usr	Код инициализации пространства пользователя (initramfs)
virt	Инфраструктура виртуализации

Ряд файлов, находящихся в корне дерева исходных кодов, также заслуживают отдельного внимания. Файл COPYING — это лицензия ядра (GNU GPL v2). Файл CREDITS — это список разработчиков, которые внесли большой вклад в разработку ядра. Файл MAINTAINERS — список людей, которые занимаются поддержкой подсистем и драйверов ядра. И наконец, Makefile — это основной файл построения ядра.

Сборка ядра

Сборка ядра достаточно проста. Это может показаться удивительным, но она даже более проста, чем компиляция и инсталляция других системных компонентов, таких, как, например, библиотеки `glibc`. В ядрах серии 2.6 встроена новая система конфигурации и построения, которая позволяет сделать эту задачу еще проще и является долгожданным улучшением по сравнению с ранними выпусками ядер.

Конфигурирование ядра

Поскольку исходный код ядра Linux доступен, то это означает, что есть возможность сконфигурировать ядро перед компиляцией. В результате у вас появляется возможность скомпилировать в ядро поддержку только необходимых драйверов и функций. Конфигурирование ядра — необходимый этап перед его компиляцией. Поскольку в ядре существует бесчисленное количество функций и вариантов поддерживаемого аппаратного обеспечения, возможностей по конфигурации, мягко говоря, *много*. Процесс конфигурации выполняется с помощью указания ряда параметров конфигурации в виде `CONFIG_`*`ВОЗМОЖНОСТЬ`*. Например, поддержка симметричной многопроцессорной обработки (Symmetric multiprocessing, SMP) устанавливается с помощью параметра `CONFIG_SMP`. Если этот параметр установлен, то поддержка функций SMP включена. Если не установлен, то функции поддержки SMP отключены. Параметры конфигурации используются как для определения того, какие файлы должны быть скомпилированы во время сборки ядра, так и для управления процессом компиляции через директивы препроцессора.

Параметры конфигурации, которые управляют процессом сборки ядра, бывают двух типов: логические (*boolean*) и с тремя состояниями (*tristate*). Логические параметры могут принимать значение *`yes`* или *`no`*. Как правило, параметры конфигурации ядра наподобие `CONFIG_PREEMPT` являются логическими. Параметры конфигурации с тремя состояниями могут принимать значение *`yes`*, *`no`* или *`module`*. Значение *`module`* означает, что данный параметр конфигурации установлен, но соответствующий ему код должен быть скомпилирован в виде модуля (т.е. как отдельный объект, который загружается динамически). В случае параметров конфигурации с тремя состояниями значение *`yes`* явно указывает на то, что соответствующий код должен быть скомпилирован в основной файл образа ядра, а не в виде модуля. Драйверы устройств обычно определяются параметрами конфигурации с тремя состояниями.

Параметры конфигурации могут также задаваться в виде строк символов или чисел. Эти параметры не управляют процессом сборки ядра, а позволяют указать значения, которые встраиваются в исходный код с помощью директив препроцессора. Например, с помощью параметра конфигурации можно указать размер статически размещаемого массива.

Ядра, которые включаются в поставки ОС Linux такими производителями, как Canonical (для Ubuntu) или Red Hat (для Fedora), компилируются как часть дистрибутива. В таких ядрах обычно имеется большой набор различных функций и практически полный набор всех драйверов устройств в виде загружаемых модулей. Это позволяет получить хорошее базовое ядро и поддержку широкого спектра оборудования в виде дополнительных модулей. Однако в любом случае разработчикам ядра нужно компилировать собственные ядра и самим решать, какие модули включать, а какие нет.

К счастью, в ядре поддерживается несколько средств, с помощью которых можно выполнить конфигурацию. Самое простое средство — это текстовая утилита командной строки:

```
$ make config
```

Эта утилита просматривает все параметры один за другим и интерактивно запрашивает у пользователя, какое значение соответствующего параметра установить — *yes*, *no* или *module* (для переменной с тремя состояниями). Эта операция требует *длительного* времени, и если у вас не почасовая оплата, то лучше использовать графическую утилиту на основе интерфейса *ncurses*:

```
$ make menuconfig
```

Имеется также и более удобная графическая утилита, использующая библиотеку *gtk+*:

```
$ make gconfig
```

В этих трех утилитах все параметры конфигурации делятся на категории наподобие Processor Type and Features (Типы и свойства процессора). При этом пользователь имеет возможность перемещаться по категориям, просматривать параметры конфигурации ядра и, разумеется, изменять их значения.

Приведенная ниже команда позволяет создать файл конфигурации, содержащий стандартные значения параметров для текущей аппаратной платформы.

```
$ make defconfig
```

И хотя значения этих параметров установлены достаточно произвольно (ходят слухи, что для аппаратной платформы i386 используется конфигурация Линуса), они являются хорошей отправной точкой, если ранее вам никогда не приходилось заниматься конфигурацией ядра. Чтобы ускорить процесс конфигурации, запустите эту команду, а потом проверьте, включена ли поддержка всех нужных вам аппаратных средств.

Параметры конфигурации сохраняются в файле `.config`, который находится в корне дерева каталогов исходного кода ядра. Большинство разработчиков считают, что гораздо проще непосредственно отредактировать этот файл конфигурации. В самом деле, с помощью любого текстового редактора достаточно легко выполнить поиск нужного параметра в этом файле и изменить его значение. После внесения изменений в файл конфигурации или при использовании существующего файла конфигурации для нового дерева каталогов исходного кода ядра его необходимо ратифицировать и обновить конфигурацию с помощью команды

```
$ make oldconfig
```

Эту команду вам также нужно запустить перед сборкой ядра.

Параметр конфигурации `CONFIG_IKCONFIG_PROC` позволяет сохранить все текущие параметры конфигурации ядра в виде сжатого файла `config.gz` и поместить его в каталог `/proc`. Это позволит вам легко создать клон текущей конфигурации при сборке нового ядра. Если этот параметр конфигурации установлен в вашем текущем ядре, то просто скопируйте файл `config.gz` из каталога `/proc` и воспользуйтесь им для создания нового ядра, как показано ниже.

```
$ zcat /proc/config.gz > .config
$ make oldconfig
```

После того как будет сконфигурировано ядро (как бы там ни было, вы должны были сделать это!), можно выполнить сборку с помощью команды

```
$ make
```

В отличие от предыдущих серий ядер, в серии 2.6 больше нет необходимости перед сборкой ядра выполнять команду `make dep`, так как дерево зависимостей создается автоматически. Также больше не нужно указывать конкретный тип сборки, например *bzImage*,

или отдельно собирать модули, как делалось в более ранних версиях. Стандартное правило, записанное в файле `Makefile`, в состоянии обработать все!

Уменьшение количества выводимых сообщений

Существует один прием, который поможет уменьшить количество сообщений, выводимых на экран во время сборки ядра, и в то же время позволит увидеть все предупреждения компилятора и сообщения об ошибках. Он заключается в перенаправлении стандартного устройства вывода команды `make` в файл, как показано ниже.

```
$ make > ../detritus
```

Если вам понадобится проанализировать выводимые сообщения, откройте этот файл в любом текстовом редакторе. Поскольку предупреждения и сообщения об ошибках по умолчанию перенаправляются на стандартное устройство вывода ошибок и отображаются на экране терминала, в этой процедуре нет особого смысла. Поэтому я выполняю приведенную ниже команду.

```
$ make > /dev/null
```

В результате весь огромный объем бессмысленных сообщений будет перенаправлен в "большую сточную канаву" `/dev/null`.

Порождение нескольких параллельных задач сборки

Программа `make` позволяет разбить процесс сборки ядра на несколько параллельно выполняющихся *заданий*. Каждое из этих заданий выполняется независимо от остальных и одновременно с остальными, что существенно ускоряет процесс сборки ядра на многопроцессорных системах. Кроме того, параллельная сборка позволяет также более эффективно использовать центральный процессор компьютера, поскольку при компиляции большого дерева исходного кода он значительное время простаивает, ожидая завершения операций ввода-вывода.

По умолчанию утилита `make` запускает только одну задачу, так как часто файлы сборки пользователей содержат некорректную информацию о зависимостях. При этом несколько заданий могут начать "наступать друг другу на пятки", что приведет к ошибкам в процессе построения. Разумеется, в файле сборки ядра `Makefile` таких ошибок нет, поэтому порождение нескольких задач не приведет к ошибкам построения. Для построения ядра с использованием нескольких параллельных задач сборки выполните команду

```
$ make -jn
```

где число *n* определяет количество порождаемых заданий. Как правило, на один процессор запускается одно или два задания. Например, на 16-процессорной машине можно воспользоваться приведенной ниже командой.

```
$ make -j32 > /dev/null
```

Используя такие великолепные утилиты, как `distcc` и `ccache`, можно еще больше ускорить время сборки ядра.

Инсталляция нового ядра

После завершения процесса сборки ядра его нужно инсталлировать в системе. Процесс инсталляции существенно зависит от используемой аппаратной платформы и типа системного загрузчика. Для того чтобы узнать, в какой каталог должен быть скопирован

образ ядра и как сделать его загружаемым, обратитесь к руководству по используемому системному загрузчику. При этом всегда сохраняйте копии одного-двух старых ядер, которые гарантированно работоспособны! Это на случай, если новое ядро не будет нормально работать или вовсе откажется загружаться.

Например, для платформы x86 при использовании системного загрузчика `grub` скопируйте загружаемый образ ядра из файла `arch/i386/boot/bzImage` в каталог `/boot` и переименуйте его в, скажем, `vmlinuz-`*`версия`*. После этого отредактируйте файл `/boot/grub/grub.conf`, добавив в него новую запись, которая соответствует новому ядру. В системах, где используется загрузчик LILO, необходимо соответственно отредактировать файл `/etc/lilo.conf` и запустить утилиту `lilo`.

К счастью, инсталляция модулей ядра автоматизирована и не зависит от аппаратной платформы. Для этого просто запустите следующую команду с правами пользователя `root`:

```
% make modules_install
```

В результате все скомпилированные модули будут инсталлированы в их корректные домашние каталоги, находящиеся в иерархии каталога `/lib/modules`.

В процессе построения ядра в корневом каталоге дерева исходного кода также создается файл `System.map`, в котором содержится таблица соответствия символов ядра их начальным адресам в памяти. Эта таблица используется при отладке для перевода адресов памяти в имена функций и переменных.

Отличия от обычных приложений

Ядро имеет некоторые уникальные отличительные особенности по сравнению с обычными пользовательскими приложениями. И хотя эти отличия не всегда приводят к усложнениям при разработке кода ядра по сравнению с пользовательскими приложениями, все же они делают сам процесс разработки немного *другим*.

Эти отличительные особенности делают ядро совершенно непохожим ни на что. Некоторые из привычных вещей в нем немного видоизменены, другие — полностью обновлены. Несмотря на то что часть различий очевидна (все знают, что ядро может делать все, что пожелает), другие различия не так очевидны. Наиболее важные отличия описаны ниже.

- Ядро не имеет доступа к библиотеке функций и к стандартным заголовкам языка C.
- Код ядра написан с использованием компилятора GNU C.
- В ядре нет такой защиты памяти, как в режиме пользователя.
- В ядре нельзя также просто использовать вычисления с плавающей точкой, как в пользовательских приложениях.
- Ядро использует стек небольшого фиксированного размера для каждого процесса.
- Поскольку обработка прерываний в ядре выполняется асинхронно, в нем реализован режим приоритетного планирования. Поддержка симметричной многопроцессорной обработки (SMP) привела к тому, что в ядре необходимо учитывать наличие параллелизма и использовать синхронизацию.
- Переносимость кода ядра очень важна.

Рассмотрим более детально перечисленные выше пункты, поскольку все разработчики ядра должны постоянно иметь их в виду.

Отсутствие библиотеки `libc` и стандартных заголовков

В отличие от обычных пользовательских приложений, ядро не компонуется со стандартной библиотекой функций языка C (и ни с какой другой библиотекой подобного типа). На то есть несколько причин, включая некоторые ситуации с дилеммой о курице и яйце, однако первопричина всему — скорость выполнения и объем кода. Полная библиотека функций языка C, и даже только самая необходимая ее часть, очень большая и неэффективная с точки зрения ядра.

Однако этот факт не должен приводить вас в замешательство, поскольку многие из функций библиотеки языка C реализованы непосредственно в ядре. Например, обычные функции работы со строками находятся в файле `lib/string.c`. Чтобы воспользоваться ими, нужно лишь включить в исходный код заголовочный файл `<linux/string.h>`.

> **Заголовочные файлы**
>
> Обратите внимание на то, что под термином *заголовочные файлы*, используемом в этой книге, имеются в виду заголовочные файлы ядра, принадлежащие его дереву исходного кода. В файлы исходного кода ядра нельзя включать заголовочные файлы, расположенные вне этого дерева каталогов, а также использовать внешние библиотеки.
>
> Основные файлы находятся в подкаталоге `include/`, расположенном в корне дерева исходного кода ядра. Например, заголовочный файл `<linux/inotify.h>` расположен в подкаталоге `include/linux/inotify.h` дерева исходных кодов.
>
> Набор заголовочных файлов, зависимых от аппаратной платформы, расположен в подкаталоге `arch/<`*платформа*`>/include/asm` дерева исходных кодов ядра. Например, при сборке ядра для платформы x86 все аппаратно-зависимые заголовочные файлы находятся в подкаталоге `arch/x86/include/asm`. Для включения этих файлов в исходный код используется префикс `asm/`, например `<asm/ioctl.h>`.

Что касается отсутствующих функций, то самая известная из них — `printf()`. Ядро не имеет доступа к функции `printf()`, однако в нем реализована функция `printk()`, которая работает ничем не хуже своего аналога. Функция `printk()` копирует отформатированную строку в буфер системных сообщений ядра (kernel log buffer), который обычно считывается с помощью программы `syslog`. Использование этой функции аналогично использованию `printf()`:

```
printk("Привет, всем! Строка '%s' и целое число '%d'\n", str, i);
```

Одно важное отличие между функциями `printf()` и `printk()` состоит в том, что в функции `printk()` можно использовать флаг важности вывода (priority flag). Этот флаг используется программой `syslog` для того, чтобы определить, где именно нужно отображать сообщения ядра. Ниже приведен пример использования этого флага.

```
printk(KERN_ERR "this is an error!\n");
```

Обратите внимание на то, что между `KERN_ERR` и выводимым сообщением отсутствует запятая. Это сделано преднамеренно. Флаг важности вывода определен с помощью директивы препроцессора как строковый литерал, который во время компиляции автоматически добавляется к выводимому сообщению. Функция `printk()` будет использоваться на протяжении всей книги.

Компилятор GNU C

Как и все "уважающие себя" ядра Unix, ядро Linux написано на языке C. Может быть, это покажется странным, но ядро Linux написано не на чистом языке C стандарта ANSI C. Там, где это только возможно, разработчики ядра используют различные расширения языка, реализованные в компиляторе *gcc* (GNU Compiler Collection — это набор компиляторов GNU, в котором содержится компилятор C, используемый для компиляции ядра, а также всего остального программного обеспечения, написанного на языке C для системы Linux).

Разработчики ядра используют как расширения языка ISO C99[1], так и расширения GNU C. Все это привело к тому, что ядро Linux было привязано к компилятору `gcc`. Несмотря на это, для компиляции ядра Linux можно также использовать любой другой современный компилятор, например Intel C, в котором достаточно хорошо реализована поддержка функций компилятора `gcc`. Самой старой поддерживаемой версией компилятора `gcc` является 3.2, однако настоятельно рекомендуется использовать версию 4.4 или более позднюю. В исходном коде ядра Linux не используются какие-либо особенные расширения стандарта C99. Кроме того, поскольку стандарт C99 является официальной редакцией языка C, эти расширения постепенно начинают использоваться и в других частях кода. Более интересные и, возможно, менее знакомые отклонения от стандарта языка ANSI C связаны с расширениями GNU C. Рассмотрим некоторые наиболее интересные расширения, которые могут встретиться в исходном коде ядра. Именно они отличают исходный код ядра от исходного кода других программных проектов, в которых, возможно, вам приходилось участвовать.

Встраиваемые функции

В стандарте C99 определены так называемые *встраиваемые функции* (inline functions), которые поддерживает компилятор GNU C. Как следует из названия, исполняемый код функции автоматически помещается компилятором во все места программы, где указан вызов этой функции. Это позволяет избежать дополнительных потерь времени процессора, связанных с вызовом функции и возвратом из нее (сохранение и восстановление регистров), а также позволяет потенциально повысить уровень оптимизации программы, поскольку компилятор теперь может одновременно оптимизировать коды и вызывающей, и вызываемой функции. Обратной стороной такой подстановки (ничто в этой жизни не дается даром!) является увеличение объема кода, увеличение используемой памяти и уменьшение эффективности использования процессорной кеш-памяти команд. Поэтому разработчики ядра используют подстановку для небольших функций, критичных ко времени выполнения. Настоятельно не рекомендуется использовать эту же возможность для больших функций, особенно когда они вызываются больше одного раза или не слишком критичны ко времени выполнения.

Встраиваемые функции объявляются с помощью ключевых слов `static` и `inline`, которые указываются при определении функции, как показано ниже.

```
static inline void wolf(unsigned long tail_size);
```

[1] Стандарт ISO C99 — это последняя основная версия стандарта ISO C. По сравнению с предыдущей редакцией стандарта ISO C90 стандарт C99 был существенно расширен и дополнен. Так, в нем появились *выделенные инициализаторы* (*designated initializers*), позволяющие выполнять поименную инициализацию отдельных полей структур, массивы переменной длины, комментарии в стиле языка C++, типы `long long` и `complex`. Тем не менее при написании ядра Linux используется только ограниченный набор возможностей стандарта C99.

Объявление функции должно предшествовать ее первому вызову, иначе компилятор не сможет выполнить подстановку. Обычно встраиваемые функции определяются в заголовочных файлах. Поскольку функция объявляется как статическая (`static`), экспортируемая функция при этом не создается. Если встраиваемая функция используется только в одном файле, то она может быть размещена в верхней части этого файла.

При написании кода ядра следует отдавать преимущество встраиваемым функциям, а не использовать вместо них сложные макросы. Это требование связано с читабельностью кода и его надежностью.

Встроенный ассемблер

Компилятор `gcc` позволяет встраивать инструкции языка ассемблера в обычные функции языка С. Эта возможность, конечно, должна использоваться только в тех частях ядра, которые уникальны для определенной аппаратной платформы.

Для встраивания ассемблерного кода используется директива компилятора `asm()`. Например, приведенная ниже встроенная директива ассемблера выполняет команду `rdtsc` процессора x86, которая возвращает значение счетчика времени (TSC — Time Stamp Counter) работы процессора, прошедшего с момента последнего сброса.

```
unsigned int low, high;
asm volatile("rdtsc" : "=a" (low), "=d" (high));
/* Переменные low и high теперь содержат значение старших и младших
   32-разрядов 64-разрядного регистра tsc */
```

Ядро Linux написано на смеси языков ассемблера и С. Язык ассемблера используется в низкоуровневых подсистемах и на участках кода, где нужна большая скорость выполнения. Бо́льшая часть кода ядра написана на языке программирования С.

Комментирование ветвлений

Компилятор `gcc` имеет встроенные директивы, позволяющие оптимизировать различные ветви условных операторов, выполнение которых наиболее или наименее вероятно. Компилятор использует эти директивы для оптимизации кода соответствующих веток. В ядре эти директивы заключены в макросы `likely()` и `unlikely()`, которые легко использовать.

Например, рассмотрим приведенный ниже оператор `if`.

```
if (error) {
   /* ... */
}
```

Для того чтобы отметить эту ветку как крайне маловероятную (т.е. управление которой вряд ли когда-либо будет передано), необходимо указать следующее:

```
/* Мы предполагаем, что переменная 'error' практически всегда
   будет равна нулю... */
if (unlikely(error)) {
    /* ... */
}
```

И наоборот, чтобы отметить приведенную ниже ветку как наиболее вероятную, нужно записать так:

```
/* Мы предполагаем, что переменная 'success' практически
   всегда не будет равна нулю... */
if (likely(success)) {
   /* ... */
}
```

Эти директивы необходимо использовать только в том случае, когда направление ветвления с большой вероятностью известно *априори* или когда необходима оптимизация какой-либо части кода за счет другой части. Важно помнить, что эти директивы увеличивают производительность, когда направление ветвления предсказано правильно, однако приводят к *потере* производительности при неправильном предсказании. Чаще всего директивы `unlikely()` и `likely()` используются для проверки ошибок, как показано выше в примерах. Как вы уже могли догадаться, в коде ядра чаще всего используется директива `unlikely()`, поскольку обычно в операторе `if` проверяется условие наступления какого-либо особого случая.

Отсутствие защиты памяти

Когда прикладная программа предпринимает незаконную попытку обращения к памяти, ядро может перехватить эту ошибку, отправить приложению сигнал `SIGSEGV` и аварийно завершить соответствующий пользовательский процесс. Если же ядро предпринимает попытку некорректного обращения к памяти, то результаты могут быть менее контролируемы. В конце концов, кто может контролировать само ядро? Нарушение правил доступа к памяти в режиме ядра приводит к ошибке *oops*, которая является наиболее часто встречающейся ошибкой ядра. Не стоит говорить, что нельзя обращаться к запрещенным областям памяти, разыменовывать указатели со значением `NULL` и так далее, однако в ядре ставки значительно выше!

Кроме того, память ядра не имеет страничной организации. Поэтому каждый сэкономленный байт памяти в ядре — это еще один байт доступной физической памяти для пользовательских приложений. Помните об этом всякий раз, когда нужно будет добавить очередную новую и полезную *функцию ядра*.

Нельзя просто использовать вычисления с плавающей точкой

Когда в пользовательской программе используются команды вычисления с плавающей точкой, ядро управляет переходом из режима работы с целыми числами в режим работы с плавающей точкой. Операции, которые ядро должно выполнить для использования команд с плавающей точкой, зависят от типа аппаратной платформы. Но обычно при этом ядро перехватывает системное прерывание, после чего инициирует переход из режима вычислений с целыми числами в режим работы с плавающей точкой.

В отличие от пользовательского приложения, в режиме ядра нет такой роскоши, как прямое использование команд с плавающей точкой, поскольку ядро не может легко перехватить системное прерывание, возникшее в режиме ядра. При использовании режима вычислений с плавающей точкой в режиме ядра, кроме прочих рутинных операций, требуется вручную сохранять и восстанавливать состояние регистров с плавающей точкой математического сопроцессора. Если говорить коротко, то лучше всего *этого никогда не делать*! За исключением редких особых случаев нельзя выполнять вычисления с плавающей точкой в режиме ядра!

Системная стек-память небольшого фиксированного размера

Пользовательские программы могут “отдохнуть” вместе со своими тоннами статически выделяемых переменных в стеке, включая структуры большого размера и многоэлементные массивы. Такое поведение является вполне законным в пользовательском режиме, поскольку область стека пользовательских программ имеет огромный размер, который

к тому же по необходимости может динамически увеличиваться в размере. Разработчики, которые писали программы под старые и не очень интеллектуальные операционные системы, как, например, MS-DOS, могут вспомнить то время, когда даже стек пользовательских программ имел фиксированный размер.

Стек, доступный в режиме ядра, не является ни большим, ни динамически изменяемым, он мал по объему и имеет фиксированный размер. Размер стека зависит от аппаратной платформы. Для платформы x86 размер стека конфигурируется на этапе компиляции и может быть равен 4 или 8 Кбайт. Исторически так сложилось, что размер стека ядра равен двум страницам памяти, что соответствует 8 Кбайт для 32-разрядных аппаратных платформ и 16 Кбайт — для 64-разрядных. Этот размер фиксирован. Каждый процесс получает свою область стека.

В следующих главах мы обсудим системный стек более подробно.

Синхронизация и параллельное выполнение

В ядре постоянно возникают конфликты из-за доступа к системным ресурсам. В отличие от однопоточной пользовательской программы, некоторые задачи ядра предполагают одновременный доступ к общим системным ресурсам, из-за чего могут возникать конфликтные ситуации. Для их разрешения используется механизм синхронизации. В частности, возможны следующие ситуации.

- В ядре Linux поддерживается приоритетный мультипрограммный режим работы. Алгоритм планирования процессов выполняется особым модулем ядра, который называется системным планировщиком (scheduler). Поэтому ядро должно осуществлять синхронное выполнение этих задач.
- В ядре Linux поддерживается симметричная многопроцессорная обработка. Поэтому, без всякого предупреждения, код ядра, выполняемый одновременно на двух или более процессорах, может одновременно обратиться к одному и тому же системному ресурсу.
- Прерывания возникают асинхронно по отношению к исполняемому коду. Поэтому, без всякого предупреждения, прерывания могут возникнуть во время обращения к ресурсу общего доступа, кроме того, обработчик прерывания также может обратиться к этому же ресурсу.
- Ядро Linux работает в мультипрограммном режиме. Поэтому, без всякого предупреждения, планировщик может прервать текущую исполняемую задачу ядра и переключиться на выполнение любой другой задачи ядра, которая также может обратиться к некоторому общему ресурсу.

Стандартное решение для предотвращения конфликтных ситуаций за ресурсы состоит в использовании спин-блокировок и семафоров. Более подробное обсуждение вопросов синхронизации и параллелизма приведено в следующих главах.

Переносимость — это важно

При разработке пользовательских программ переносимость *не всегда* ставится во главу угла, однако операционная система Linux является переносимой и должна оставаться таковой. Это означает, что аппаратно-независимый код, написанный на языке C, должен компилироваться без ошибок и правильно выполняться на большом количестве

систем. С другой стороны, аппаратно-зависимый код должен быть корректно разнесен по соответствующим каталогам дерева исходных кодов ядра.

Несколько правил, таких как не создавать зависимости от порядка следования байтов, обеспечивать возможность использования кода на 64-разрядных системах, не привязываться к размеру страницы памяти или машинного слова и другие, имеют большое значение. Вопросы переносимости более подробно рассматриваются в одной из следующих глав.

Резюме

Как вы уже поняли, ядро Linux обладает уникальными качествами. В нем принят ряд особых правил и исключений, которые являются очень важными и влияют на работу всей системы. Выше мы уже говорили, что сложность кода ядра и необходимый уровень подготовки программистов для его разработки принципиально ничем не отличаются от любого другого большого программного проекта. Самым важным шагом на пути к разработке ядра Linux является осознание того, что ядро не так уж и страшно, как оно кажется на первый взгляд. Незнакомо? Конечно! Непреодолимо? Не для всех!

Вводный материал, который был представлен в первой главе, и базовые моменты, описанные в текущей, надеюсь, станут хорошим фундаментом для тех знаний, которые будут получены при прочтении всей книги. В каждой из последующих глав будут описаны конкретные подсистемы ядра и принципы их работы. Помимо всего прочего, очень важно, чтобы вы самостоятельно смогли разобраться в исходном коде ядра и внести в него изменения. Только в процессе реального знакомства с исходным кодом и выполнения экспериментов с ним вы сможете все понять. В конце концов, исходный код доступен совершенно свободно! Так пользуйтесь этой возможностью!