http://www.ibm.com/developerworks/ru/library/l-linux_kernel_01/index.html
Разработка модулей ядра Linux

Часть 1. Первые шаги

Олег Цилюрик
Опубликовано 01.01.2012

Серия контента:

Этот контент является частью серии:**Разработка модулей ядра Linux**

Эта статья является первой в протяжённом цикле статей, который в последствии должен превратиться в **пошаговое руководство** по написанию модулей ядра Linux. Но для того, чтобы такая амбициозная задача имела шансы на успех, а её результат был измеримым и конкретным, потребуется с самого начала разграничить области интересов и выделить зоны ответственности при разработке целевых модулей. В данном цикле не будут рассматриваться следующие вопросы.

- детали внутреннего устройства и функционирования ядра Linux, а также выявляющиеся в нём проблемы. Эти вопросы относятся к сфере компетенции команды разработчиков ядра, возглавляемой Линусом Торвальдсом.
- утилиты и библиотеки, поставляемые в составе Linux. Разработчику достаточно уметь использовать эти инструменты при построении модулей ядра, но более глубокие знания — это уже прерогатива сообществ GNU, FSF и разработчиков независимых проектов.
- вопросы интеграции создаваемого модуля в дерево исходных кодов Linux или любого его дистрибутива. Эти вопросы должны решаться системотехниками или внедренцами, берущими на себя ответственность за дальнейшую судьбу разрабатываемого проекта. Поэтому демонстрируемые примеры модулей будут создаваться не в дереве исходных кодов системы, а в отдельных каталогах целевых проектов.

Разработчика модулей ядра должны интересовать исключительно вопросы создания собственного модуля (драйвера) для использования в рамках некого целевого прикладного программного проекта. Поэтому все вопросы, выходящие за рамки этой практической задачи, не будут рассматриваться в рамках данного цикла статей.

Эта короткое, но необходимое, введение поясняет, что можно ожидать от этой и последующих статей, а на какие вопросы они не смогут ответить.

Создание первого модуля ядра

Лучший способ научиться плавать — начать плавать. Поэтому, вместо скучных обсуждений терминологии, систематизации и архитектуры цикл начинается сразу с написания кода модулей ядра. Такой код будет интуитивно понятен любому программисту без особых пояснений. Хотя в дальнейшем, конечно, обязательно

придётся вернуться к обсуждению скучных вещей в объёме, необходимом для достижения поставленной цели. Обычно любую иллюстрацию из мира программирования начинают с примера "Hello world!". На самом деле, "Hello world!" модуль - это не самый лучший пример, так как все пишущие о ядре авторы используют именно его. Вместо этого лучше создать несколько простейших модулей, в конечном итоге имеющих суммарно такую же потребительскую ценность, как и "Hello world!", но при этом сразу иллюстрирующих гораздо больше понятий из мира модулей ядра и открывающих дорогу для дальнейших экспериментов. В листинге 1 приведен код вызываемого модуля (файл **md1.c**), а в листинге 2 — код вызывающего модуля (файл **md2.c**)

Листинг 1. Вызываемый модуль ядра Linux

```
1
2
3     #include <linux/init.h>
4     #include <linux/module.h>
5     #include "md.h"
6     MODULE_LICENSE( "GPL" );
7     MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
8     char* md1_data = "Привет мир!";
9     extern char* md1_proc( void ) {
10       return md1_data;
11    }
12    static char* md1_local( void ) {
13       return md1_data;
14    }
15    extern char* md1_noexport( void ) {
16       return md1_data;
17    }
18    EXPORT_SYMBOL( md1_data );
19    EXPORT_SYMBOL( md1_proc );
20    static int __init md_init( void ) {
21       printk( "+ module md1 start!\n" );
22       return 0;
23    }
24    static void __exit md_exit( void ) {
25       printk( "+ module md1 unloaded!\n" );
26    }
27    module_init( md_init );
28    module_exit( md_exit );
29
30
```

Листинг 2. Вызывающий модуль ядра Linux

```
1
2     #include <linux/init.h>
3     #include <linux/module.h>
4     #include "md.h"
5     MODULE_LICENSE( "GPL" );
6     MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
7     static int __init md_init( void ) {
8        printk( "+ module md2 start!\n" );
9        printk( "+ data string exported from md1 : %s\n", md1_data );
10       printk( "+ string returned md1_proc() is : %s\n", md1_proc() );
11       return 0;
12    }
13    static void __exit md_exit( void ) {
14       printk( "+ module md2 unloaded!\n" );
15    }
16    module_init( md_init );
17    module_exit( md_exit );
18
```

```
16
```

Также все модули, включают заголовочный файл (только обеспечивающий их синтаксическую связь), приведенный ниже:

```
1   extern char* md1_data;
2   extern char* md1_proc( void );
```

Как и для сборки любого проекта, в данном случае понадобится файл сценария сборки с именем **Makefile**, приведенный в листинге 3. Опытные пользователи легко поймут, что в нем собираются 3 независимых модуля.

Листинг 3. Типовой сценарий сборки

```
1
2    CURRENT = $(shell uname -r)
3    KDIR = /lib/modules/$(CURRENT)/build
4    PWD = $(shell pwd)
5    TARGET1 = md1
6    TARGET2 = md2
7    TARGET3 = md3
8    obj-m   := $(TARGET1).o $(TARGET2).o $(TARGET3).o
9    default:
10           $(MAKE) -C $(KDIR) M=$(PWD) modules
11   clean:
12           @rm -f *.o .*.cmd .*.flags *.mod.c *.order
13           @rm -f .*.*.cmd *~ *.*~ TODO.*
14           @rm -fR .tmp*
15           @rm -rf .tmp_versions
16   disclean: clean
17           @rm *.ko *.symvers
18
```

Примечание. Согласно синтаксическим правилам утилиты `make`, каждая строка, имеющая отступ в листинге 3 и не начинающаяся с первого символа строки, должна начинаться с одного или нескольких символов TAB, но ни в коем случае не с пробелов.

Более детально разнообразные варианты содержимого **Makefile** для различных случаев сборки модулей будут рассмотрены позже, но и представленной информации уже достаточно, чтобы успешно выполнить команду сборки.

Листинг 4. Результат успешной сборки модуля

```
1
2    $ make
3    make -C /lib/modules/2.6.32.9-70.fc12.i686.PAE/build M=/home/olej/TMP modules
4    make[1]: Entering directory `/usr/src/kernels/2.6.32.9-70.fc12.i686.PAE'
5      CC [M]  /home/olej/TMP/md1.o
6    /home/olej/TMP/md1.c:14: предупреждение: ‘md1_local’ определена, но нигде не исп
7      CC [M]  /home/olej/TMP/md2.o
8      CC [M]  /home/olej/TMP/md3.o
9      Building modules, stage 2.
10     MODPOST 3 modules
11     CC      /home/olej/TMP/md1.mod.o
12     LD [M]  /home/olej/TMP/md1.ko
13     CC      /home/olej/TMP/md2.mod.o
14     LD [M]  /home/olej/TMP/md2.ko
15     CC      /home/olej/TMP/md3.mod.o
16     LD [M]  /home/olej/TMP/md3.ko
17   make[1]: Leaving directory `/usr/src/kernels/2.6.32.9-70.fc12.i686.PAE'
18
```

В листинге 4 приведен пример успешной сборки модуля (предупреждение касательно строки 14 файла **md1.c** не существенно), результаты сборок других модулей должны выглядеть аналогично. На этом вся работа по программированию и сборке завершена, и можно переходить к дальнейшим

экспериментам и наблюдениям. Все ключевые понятия и термины, возникающие в ходе этих экспериментов, имеющие существенное значение для понимания природы и техники модулей, буду выделяться **таким шрифтом**.
В результате сборки были созданы три файла модуля ядра:

```
$ ls *.ko
md1.ko  md2.ko  md3.ko
```

Это файлы объектного формата компилятора gcc, расширенные некоторыми дополнительными символами (в терминологии объектных модулей):

```
$ file md1.ko
md1.ko: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

Попробуем инсталлировать (загрузить) один из новых модулей в системе:

```
$ sudo insmod md2.ko
insmod: error inserting 'md2.ko': -1 Unknown symbol in module
```

Возникла ошибка, потому что модуль **содержит** ссылки на неизвестные ядру имена (хотя известно, что эти недостающие имена определены в другом модуле md1). В системном журнале об этой ошибке будет выведена следующая информация:

```
$ dmesg | tail -n30 | grep md
md2: Unknown symbol md1_data
md2: Unknown symbol md1_proc
```

Следует выполнить загрузку модулей в другом, на этот раз правильном, порядке:

```
$ sudo insmod md1.ko<br>$ sudo insmod md2.ko<br>$ lsmod | grep md
md2                    646  0
md1                    860  1 md2
```

Всё прошло нормально, но в разработанных модулях присутствовали вызовы функции `printk()`, до сих пор не рассматривавшейся, но, по аналогии с `printf()` она должна была выводить текстовые сообщения по ходу загрузки модулей. Всё это так, но в **пространстве пользователя**, при запуске приложения и вызове `printf()` вывод осуществляется на **управляющий терминал**, а таким терминалом является текстовая консоль или приложение графического терминала, если выполнение происходит в среде X Window System. Загрузка модулей в свою очередь выполнялась в пространстве ядра, где нет и не может быть никакого управляющего терминала, поэтому вывод `printk()` направляется демону системного журналирования, который помещает его, в частности, в **системный журнал** (**/var/log/messages**). Этот вопрос еще будет обсуждаться дальше, а пока достаточно просто воспользоваться командой чтения системного журнала (dmesg), как показано ниже:

```
$ dmesg | tail -n60 | grep +
+ module md1 start!
+ module md2 start!
+ data string exported from md1 : Привет мир!
+ string returned md1_proc() is : Привет мир!
```

Другой способ найти выведенные сообщения, это изучить файл системного журнала, но в некоторых дистрибутивах для этого могут потребоваться права **root**:

```
$ sudo cat /var/log/messages | tail -n150 | grep +
Dec 17 20:08:03 notebook kernel: + module md1 start!
Dec 17 20:08:09 notebook kernel: + module md2 start!
Dec 17 20:08:09 notebook kernel: + data string exported from md1 : Привет мир!
Dec 17 20:08:09 notebook kernel: + string returned md1_proc() is : Привет мир!
```

После успешного создания и загрузки модулей можно их выгрузить с помощью команды `rmmod`:

```
$ sudo rmmod md1
ERROR: Module md1 is in use by md2
```

Однако на этом шаге снова возникает ошибка. Рассмотрим листинг выполнения команды `lsmod`, расположенный несколькими абзацами выше:

- на модуль **md1** ссылается некоторые другие модули или объекты ядра: цифра **1** — это число таких ссылающихся модулей, которое называется **счётчиком ссылок**;
- дальше за счётчиком ссылок указывается **список** тех модулей, откуда исходят такие ссылки, в данном случае, это один модуль **md2**;
- до тех пор, пока число ссылок на любой модуль в системе **не станет нулевым**, модуль **не может быть выгружен**;
- другими словами, модуль может быть выгружен только после того, как будут выгружены все ссылающиеся на него другие модули (загруженные **после** него — здесь не может возникнуть цикличности или перекрёстности ссылок);
- бывают случаи (как будет показано дальше), когда модуль вообще не может быть выгружен, в частности, когда счётчик ссылок для модуля не может быть сделан нулевым в силу каких-то причин.

Следующая попытка выгрузить модули уже учитывает эти правила, сначала выгружая модуль **md2**, и только потом **md1**:

```
$ sudo rmmod md2<br>$ sudo rmmod md1<br>$ lsmod | grep md
$
```

К этому моменту модули были созданы, загружены и использованы (по крайней мере, удалось обнаружить их диагностический вывод), а затем выгружены, проделав тем самым полный жизненный цикл. В конечном итоге операционная система вернулась в **исходное состояние**. Теперь можно сделать некоторые комментарии к отдельным фрагментам исходного кода модулей, которые могут показаться не совсем очевидными:

1. Модуль **md1экспортирует** для использования другими модулями имя процедуры `md1_proc()` и, что далеко не так очевидно, имя структуры данных `md1_data`. Любой другой модуль (**md2**) может использовать в своём коде любые **экспортируемые** имена. Это могут быть имена, экспортируемые ранее загруженными модулями, но гораздо чаще это имена, **экспортируемые ядром**. Это множество экспортируемых имён ядра далее будет называться **API ядра**. Примером одного из вызовов из набора API ядра в показанных фрагментах кода является вызов `printk()`.
2. Модуль **md2**, использующий экспортируемое имя, связывается с этим именем по прямому **абсолютному адресу**. Как следствие этого, любые изменения (новая сборка), вносимые в ядро или экспортирующие модули, делают собранный модуль непригодным для использования. Именно поэтому бессмысленно предоставлять модуль в собранном виде — он должен собираться только на месте использования.
3. Модуль сможет использовать только те имена, которые явно экспортированы. В модуле **md1** специально показаны два других имени: `md1_local()` является локальным именем (модификатор `static`), непригодным для связывания, а имя `md1_noexport()` не объявлено как экспортируемое имя и также не может быть использовано вне модуля.
4. Почему в качестве строки, выводимой `md1_proc()`, была выбрана строка "Привет мир!" в русскоязычном написании? Для того, чтобы сразу проверить прозрачность настроек самых разных подсистем Linux на работу с UNICODE представлением символьных данных в кодировке UTF-8 — в ранних версиях Linux всё было не так однозначно. Что в данном случае понимается под прозрачностью? Это единообразное и слаженное поведение на таких кодировках и подсистемы

клавиатуры, и отображение в графических терминалах и текстовой консоли, и поведение системного журнала. Кроме того, для большей степени общности, интересно работать со строковыми представлениями, для которых значения `strlen()` значительно больше визуально видимой длины строки.

5. Зачем каждую выводимую строку предварять строкой "+"? Это **маркер**, отмечающий вывод из **собственных** модулей. В качестве него можно выбрать любой символ или вообще отказаться от него (что чаще всего и происходит). Но если настройки Linux таковы, что работают различные сервисы аудита или подобные службы, то они могут «засыпать» системный журнал достаточно плотным потоком своих сообщений, а сообщения собственных модулей будут сильно разрежены таким потоком. Так что их придётся потом разыскивать в этом потоке. Заблаговременно предварять сообщения собственных модулей фиксированными маркерами — это простейший способ позже осуществить их отбор и группировку, по крайней мере, в иллюстрационных целях, как и было показано. Отобрать собственные сообщения можно с помощью команд, подобных приведенной ниже:

```
$ dmesg | tail -n60 | grep +
```

Существует и другой способ отобрать интересующие сообщения из системного журнала. Если знать внешний вид сообщений аудита, можно напротив: **отбраковать** засоряющие листинг сообщения по соответствующему регулярному выражению. Достаточно часто журнал может быть засорён повторяющимися сообщениями вида:

```
$ dmesg | tail -n2
audit(:0): major=340 name_count=0: freeing multiple contexts (16)
audit: freed 16 contexts
```

В этом случае используется фильтр, который в дальнейшем иногда будет фигурировать в демонстрируемых примерах:

```
$ dmesg | tail -n50 | grep -v ^audit
```

В листинге 5 приведен исходный код последнего из модулей, представленных в архиве проекта (файл **md3.c**):

Листинг 5. Модифицированная версия модуля md2

```

#include <linux/init.h>
#include <linux/module.h>
#include "md.h"
MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
static int __init md_init( void ) {
   printk( "+ module md3 start!\n" );
   printk( "+ data string exported from md1 : %s\n", md1_data );
   printk( "+ string returned md1_proc() is : %s\n", md1_proc() );
   return -1;
}
module_init( md_init );

```

Это ещё одна форма модуля, которая не часто упоминается в других статьях, посвященных модулям ядра, но очень удобная для отладочных действий с модулями и для других специальных действий:

- функция инициализации этого модуля, выполнив все предписанные ей действия, **преднамеренно** возвращает ненулевое значение, что означает ошибку инициализации модуля;
- тогда такой модуль **не будет подгружен** к ядру, но произойдёт это уже **после** выполнения кода инициализирующей функции модуля **в пространстве ядра**;

- а если такой модуль по замыслу не загружается, то он, в принципе, **может**не иметь функции выгрузки.
  Такой модуль очень напоминает привычные приложения пространства пользователя (процессы), но только код этот выполняется в пространстве ядра со всеми **ядерными** привилегиями. Ниже приведен вывода после запуска этого модуля:

```
$ sudo insmod md3.ko
insmod: error inserting 'md3.ko': -1 Operation not permitted
$ dmesg | tail -n60 | grep +
+ module md3 start!
+ data string exported from md1 : Привет мир!
+ string returned md1_proc() is : Привет мир!
$ lsmod | grep md
md1                     860  0
```

Заключение

В первой статье этого цикла был представлен краткий обзор техники модульного программирования, который можно подытожить следующими утверждениями:

- код модуля объявляет две функции: инициализации и финализации (завершения);
- имена функций, выполняющих эти задачи, объявляются в макросах `module_init()` и `module_exit()`;
- эти функции должны в точности соответствовать прототипу, показанному в листингах примеров;
- функция инициализации выполняется при загрузке модуля в ядро, если эта функция возвращает нулевое значение (успех), то код модуля остаётся резидентным и выполняет дальнейшую работу;
- функция завершения вызывается при выгрузке модуля командой `rmmod`;
- функция завершения по своему прототипу **не имеет возвращаемого значения**, поэтому, начавшись, она уже не имеет механизмов сообщить о своём неудачном выполнении;
- большинство операций над модулями (`insmod`, `rmmod`, `modprobe`) требуют прав **root**, но некоторые индикативные команды (`lsmod`) могут выполняться и от имени ординарного пользователя.
  Названные выше соглашения по объявлению функций инициализации и завершения являются общепринятыми. Но существует ещё один, не документированный, способ описания этих функций: воспользоваться их предопределёнными именами, а именно `init_module()` и `cleanup_module()`, как показано ниже:

```
int init_module( void ) {
   ...
}
void cleanup_module( void ) {
   ...
}
```

  При такой записи необходимость в использовании макросов `module_init()` и `module_exit()` отпадает, а использовать модификатор `static` с этими функциями нельзя (именно эти имена и используются при загрузке и удалении модуля). Конечно, такая запись не способствует улучшению читаемости кода, но она может существенно сократить количество рутинного кода, особенно в коротких иллюстративных примерах, так что она будет использоваться в демонстрируемых примерах.
  Ресурсы для скачивания
- этот контент в PDF

- исходный код примеров (export-data-1.tgz | 6KB)
  Похожие темы
- Инструкция по работе с примерами исходного кода в цикле статей "Разработка модулей ядра Linux"
- В разделе Linux сайта developerWorks находятся сотни практических руководств и учебных пособий, материалы для скачивания, форумы для обсуждения и множество других ресурсов для разработчиков и администраторов Linux.
- Знакомьтесь с продуктами IBM различными способами: загружайте ознакомительные версии, испытывайте продукты в онлайновом режиме или в облачной среде или проведите несколько часов в SOA Sandbox, чтобы узнать, как эффективно создавать SOA-приложения.
- Следите за публикациями developerWorks в Твиттере или подпишитесь на канал твитов по Linux на developerWorks.