Тема №7. Модули ядра

Что такое "Модуль Ядра"?

Итак, Вы хотите писать модули ядра. Вы знакомы с языком C и у вас есть опыт создания обычных программ, а теперь вы хотите забраться туда, где свершается великое таинство. Туда, где один ошибочный указатель может "снести" файловую систему или "подвесить" компьютер.

Так что же такое "модуль ядра"? Модуль -- это некий код, который может быть загружен или выгружен ядром по мере необходимости. Модули расширяют функциональные возможности ядра без необходимости перезагрузки системы. Например, одна из разновидностей модулей ядра, драйверы устройств, позволяют ядру взаимодействовать с аппаратурой компьютера. При отсутствии поддержки модулей нам пришлось бы писать монолитные ядра и добавлять новые возможности прямо в ядро. При этом, после добавления в ядро новых возможностей, пришлось бы перезагружать систему.

Как модули попадают в ядро?

Вы можете просмотреть список загруженных модулей командой **lsmod**, которая в свою очередь обращается за необходимыми сведениями к файлу `/proc/modules`.

Как же модули загружаются ядром? Когда ядро обнаруживает необходимость в тех или иных функциональных возможностях, еще не загруженных в память, то демон **kmod** [1] вызывает утилиту **modprobe**, передавая ей строку в виде:

- Название модуля, например `softdog` или `ppp`.
- Универсальный идентификатор, например `char-major-10-30`.

Если утилите **modprobe** передается *универсальный идентификатор*, то она сначала пытается отыскать имя соответствующего модуля в файле `/etc/modules.conf`, где каждому универсальному идентификатору поставлено в соответствие имя модуля, например:

```
alias char-major-10-30 softdog
```

Это соответствует утверждению: "Данному универсальному идентификатору соответствует файл модуля `softdog.ko`".

Затем **modprobe** отыскивает файл `/lib/modules/version/modules.dep`, чтобы проверить -- не нужно ли загружать еще какие-либо модули, от которых может зависеть заданный модуль. Этот файл создается командой **depmod -a** и описывает зависимости модулей. Например, модуль `msdos.ko` требует, чтобы предварительно был загружен модуль `fat.ko`. Если модуль **Б** экспортирует ряд имен (имена функций, переменных и т.п.), которые используются модулем **А**, то говорят, что "Модуль **А** зависит от модуля **Б**".

И наконец **modprobe** вызывает **insmod**, чтобы сначала загрузить необходимые, для удовлетворения зависимостей, модули, а затем и запрошенный модуль. Вызывая **insmod**, утилита **modprobe** указывает ей каталог, `/lib/modules/version/` [2] -- стандартный путь к модулям ядра. Утилита **insmod** ничего не "знает" о размещении модулей ядра, зато это "знает" утилита **modprobe**. Таким образом, если вам

необходимо загрузить модуль `msdos`, то вам необходимо дать следующие команды:

```
insmod /lib/modules/2.6.0/kernel/fs/fat/fat.ko
insmod /lib/modules/2.6.0/kernel/fs/msdos/msdos.ko
```

или просто:

```
modprobe -a msdos
```

В большинстве дистрибутивов Linux, утилиты **modprobe, insmod, depmod** входят в состав пакета **modutils** или **mod-utils**.

Прежде чем закончить эту главу, я предлагаю вкратце ознакомиться с содержимым файла `/etc/modules.conf`:

```
### This file is automatically generated by modules-update
#
# Please do not edit this file directly. If you want to change or add
# anything please take a look at the files in /etc/modules.d and read
# the manpage for modules-update.
#
### modules-update: start processing /etc/modules.d/aliases
# Aliases to tell insmod/modprobe which modules to use
path[misc]=/lib/modules/2.6.?/local
keep
path[net]=~p/mymodules
options mydriver irq=10
alias eth0 eepro
```

Строки, начинающиеся с символа "#" являются комментариями. Пустые строки игнорируются.

Строка `path[misc]` сообщает **modprobe** о том, что модули ядра из категории misc следует искать в каталоге `/lib/modules/2.6.?/local`. Как видите, здесь вполне допустимы шаблонные символы.

Строка `path[net]` задает каталог размещения модулей категории net, однако, директива **keep**, стоящая выше, сообщает, что каталог `~p/mymodules` не замещает стандартный путь поиска модулей (как это происходит в случае с `path[misc]`), а лишь добавляется к нему.

Строка `alias` говорит о том, что если запрошена загрузка модуля по универсальному идентификатору `eth0`, то следует загружать модуль `eepro.ko`

Вы едва ли встретите в этом файле строки, подобные:

```
alias block-major-2 floppy
```

поскольку **modprobe** уже знает о существовании стандартных драйверов устройств, которые используются в большинстве систем.

"Hello, World" (часть 1): Простейший модуль ядра.

Когда первый пещерный программист высекал свою первую программу для каменного компьютера, это была программа, которая рисовала "Hello, World!" поперек изображения антилопы. Древнеримские учебники по программированию начинались с программы "Salut, Mundi". Я не знаю -- что случается с людьми, которые порывают с этой традицией, но думаю, что лучше этого и не знать.

Ниже приводится исходный текст самого простого модуля ядра, какой только

возможен. Пока только ознакомьтесь с его содержимым, а компиляцию и запуск модуля мы обсудим в следующем разделе.

Пример 1. hello-1.c

```
/*
 *  hello-1.c - Простейший модуль ядра.
 */
#include <linux/module.h>       /* Необходим для любого модуля ядра */
#include <linux/kernel.h>       /* Здесь находится определение KERN_ALERT */

int init_module(void)
{
        printk("<1>Hello world 1.\n");

  /*
   * Если вернуть ненулевое значение, то это будет воспринято как признак
ошибки,
   * возникшей в процессе работы init_module; в результате модуль не будет
загружен.
   */
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_ALERT "Goodbye world 1.\n");
}
```

Любой модуль ядра должен иметь по меньшей мере хотя бы две функции: функцию инициализации модуля -- `init_module()`, которую вызывает **insmod** во время загрузки модуля, и функцию завершения работы модуля -- `cleanup_module()`, которую вызывает **rmmod**. Начиная с ядра, версии 2.3.13, требования к именованию начальной и конечной функций были сняты. Теперь вы можете давать им свои имена. Новый метод именования является более предпочтительным, однако многие по-прежнему продолжают использовать имена `init_module()` и `cleanup_module()`.

Обычно функция `init_module()` выполняет регистрацию обработчика какого-либо события или замещает какую-либо функцию в ядре своим кодом (который, как правило, выполнив некие специфические действия, вызывает оригинальную версию функции в ядре). Функция `cleanup_module()` является полной противоположностью, она производит "откат" изменений, сделаных функцией `init_module()`, что делает выгрузку модуля безопасной.

И наконец, любой модуль ядра должен подключать заголовочный файл `linux/module.h`. В нашем примере мы подключали еще один файл -- `linux/kernel.h`, но лишь для того, чтобы получить доступ к определению `KERN_ALERT`, которое более подробно будет обсуждаться в далее.

Знакомство с `printk()`

Несмотря на столь красноречивое название, функция `printk()` вовсе не предназначена для вывода информации на экран, даже не смотря на то, что мы использовали ее в своем примере именно для этой цели! Основное назначение этой функции -- дать ядру механизм регистрации событий и предупреждений. Поэтому, каждый вызов `printk()` сопровождается указанием приоритета, в нашем примере это `<1>` и `KERN_ALERT`. Всего в ядре определено 8 различных уровней приоритета

для функции `printk()` и каждый из них имеет свое макроопределение, таким образом нет необходимости писать числа, лишенные смысла (имена уровней приоритета и их числовые значения вы найдете в файле `linux/kernel.h`). Если уровень приоритета не указывается, то по-умолчанию он принимается равным `DEFAULT_MESSAGE_LOGLEVEL`.

Найдите время и просмотрите содержимое этого файла. Здесь вы найдете краткое описание значения каждого из уровней. На практике считается дурным тоном указание уровней приоритета числовым значением, например так: `<4>`. Для этих целей лучше пользоваться именами макроопределений, например: `KERN_WARNING`.

Если задан уровень ниже, чем `int console_loglevel`, то сообщение выводится на экран. Если запущены и **syslog**, и **klogd**, то сообщение попадет также и в системный журнал `/var/log/messages`, при этом оно может быть выведено на экран, а может и не выводиться. Мы использовали достаточно высокий уровень приоритета `KERN_ALERT` для того, чтобы гарантировать вывод сообщения на экран функцией `printk()`. Когда вы вплотную займетесь созданием модулей ядра, вы будете использовать уровни приоритета наиболее подходящие под конкретную ситуацию.

Сборка модулей ядра

Чтобы модуль был работоспособен, при компиляции необходимо передать **gcc** ряд опций. Кроме того, необходимо чтобы модули компилировались с предварительно определенными символами. Ранние версии ядра полностью полагались, в этом вопросе, на программиста и ему приходилось явно указывать требуемые определения в `Makefile`-ах. Несмотря на иерархическую организацию, в `Makefile`-ах, на вложенных уровнях, накапливалось такое огромное количество параметров настройки, что управление и сопровождение этих настроек стало довольно трудоемким делом. К счастью появился **kbuild**, в результате процесс сборки внешних загружаемых модулей теперь полностью интегрирован в механизм сборки ядра. Дополнительные сведения по сборке модулей, которые не являются частью официального ядра (как в нашем случае), вы найдете в файле `linux/Documentation/kbuild/modules.txt`.

А теперь попробуем собрать наш с вами модуль hello-1.c . Соответствующий `Makefile` содержит всего одну строку:

Пример 2. Makefile для модуля ядра

```
obj-m += hello-1.o
```

Для того, чтобы запустить процесс сборки модуля, дайте команду **make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules** (*если у вас в каталоге* `/usr/src` *присутствует символическая ссылка* `linux` *на каталог с исходными текстами ядра, то команда сборки может быть несколько упрощена:* **make -C /usr/src/linux SUBDIRS=$PWD modules**). На экран должно быть выведено нечто подобное:

```
[root@pcsenonsrv test_module]# make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD
modules
make: Entering directory `/usr/src/linux-2.6.x
  CC [M]  /root/test_module/hello-1.o
  Building modules, stage 2.
  MODPOST
  CC      /root/test_module/hello-1.mod.o
```

```
  LD [M]  /root/test_module/hello-1.ko
make: Leaving directory `/usr/src/linux-2.6.x
```

Обратите внимание: в ядрах версии 2.6 введено новое соглашение по именованию объектных файлов модулей. Теперь, они имеют расширение `.ko` (взамен прежнего `.o`), что отличает их от обычных объектных файлов. Дополнительную информацию по оформлению `Makefile`-ов модулей вы найдете в `linux/Documentation/kbuild/makefiles.txt`. Обязательно прочтите этот документ прежде, чем начнете углубляться в изучение `Makefile`-ов.

Итак, настал торжественный момент -- теперь можно загрузить свежесобранный модуль! Дайте команду **insmod ./hello-1.ko** (появляющиеся сообщения о "загрязнении" ядра вы сейчас можете просто игнорировать, вскоре мы обсудим эту проблему).

Любой загруженный модуль ядра заносится в список `/proc/modules`, так что дружно идем туда и смотрим содержимое этого файла. как вы можете убедиться, наш модуль стал частью ядра. С чем вас и поздравляем, теперь вы стали одним из авторов кода ядра! Вдоволь насладившись ощущением новизны, выгрузите модуль командой **rmmod hello-1** и загляните в файл `/var/log/messages`, здесь вы увидите сообщения, которые сгенерировал ваш модуль. (`/var/log/kern.log`).

Измените содержимое файла `hello-1.c` так, чтобы функция `init_module()` возвращала бы какое либо ненулевое значение и проверьте -- что получится?

Hello World (часть 2)

Как мы уже упоминали, начиная с ядра, версии 2.3.13, требования к именованию начальной и конечной функций модуля были сняты. Достигается это с помощью макроопределений `module_init()` и `module_exit()`. Они определены в файле `linux/init.h`. Единственное замечание: начальная и конечная функции должны быть определены выше строк, в которых вызываются эти макросы, в противном случае вы получите ошибку времени компиляции. Ниже приводится пример использования этих макроопределений:

Пример 3. hello-2.c

```
/*
 *  hello-2.c - Демонстрация использования макроопределений module_init() и
module_exit().
 */
#include <linux/module.h>       /* Необходим для любого модуля ядра */
#include <linux/kernel.h>       /* Здесь находится определение KERN_ALERT */
#include <linux/init.h>         /* Здесь находятся определения макросов */

static int __init hello_2_init(void)
{
        printk(KERN_ALERT "Hello, world 2\n");
        return 0;
}

static void __exit hello_2_exit(void)
{
        printk(KERN_ALERT "Goodbye, world 2\n");
}

module_init(hello_2_init);
module_exit(hello_2_exit);
```

Теперь мы имеем в своем багаже два настоящих модуля ядра. Добавить сборку второго модуля очень просто:

Пример 4. Makefile для сборки обоих модулей

```
obj-m += hello-1.o
obj-m += hello-2.o
```

Теперь загляните в файл `linux/drivers/char/Makefile`. Он может рассматриваться как пример полноценного `Makefile` модуля ядра. Здесь видно, что ряд модулей жестко "зашиты" в ядро (obj-y), но нигде нет строки obj-m. Почему? Знакомые с языком сценариев командной оболочки легко найдут ответ. Все записи вида obj-$(CONFIG_FOO) будут заменены на obj-y или obj-m, в зависимости от значения переменных CONFIG_FOO. Эти переменные вы сможете найти в файле `.config`, который был создан во время конфигурирования ядра с помощью **make menuconfig** или что-то вроде этого.

Hello World (часть 3): Макроопределения __init и __exit

Это демонстрация особенностей ядра, появившихся, начиная с версии 2.2. Обратите внимание на то, как изменились определения функций инициализации и завершения работы модуля. Макроопределение `__init` вынуждает ядро, после выполнения инициализации модуля, освободить память, занимаемую функцией, правда относится это только к встроенным модулям и не имеет никакого эффекта для загружаемых модулей. Если вы мысленно представите себе весь процесс инициализации встроенного модуля, то все встанет на свои места.

То же относится и к макросу `__initdata`, но только для переменных.

Макроопределение `__exit` вынуждает ядро освободить память, занимаемую функцией, но только для встроенных модулей, на загружаемые модули это макроопределение не оказывает эффекта. Опять же, если вы представите себе -- когда вызывается функция завершения работы модуля, то станет понятно, что для встроенных модулей она не нужна, в то время как для загружаемых модулей -- просто необходима.

Оба этих макроса определены в файле `linux/init.h` и отвечают за освобождение неиспользуемой памяти в ядре. Вам наверняка приходилось видеть на экране, во аремя загрузки, сообщение примерно такого содержания: `Freeing unused kernel memory: 236k freed`. Это как раз и есть результат работы данных макроопределений.

Пример 5. hello-3.c

```
/*
 *  hello-3.c - Использование макроопределений __init, __initdata и __exit.
 */
#include <linux/module.h>       /* Необходим для любого модуля ядра */
#include <linux/kernel.h>       /* Здесь находится определение KERN_ALERT */
#include <linux/init.h>         /* Здесь находятся определения макросов */

static int hello3_data __initdata = 3;

static int __init hello_3_init(void)
{
        printk(KERN_ALERT "Hello, world %d\n", hello3_data);
        return 0;
}
```

```
static void __exit hello_3_exit(void)
{
        printk(KERN_ALERT "Goodbye, world 3\n");
}

module_init(hello_3_init);
module_exit(hello_3_exit);
```

Hello World (часть 4): Вопросы лицензирования и документирования модулей

Если у вас установлено ядро 2.4 или более позднее, то наверняка, во время запуска примеров модулей, вам пришлось столкнуться с сообщениями вида:

```
# insmod hello-3.o
Warning: loading hello-3.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about tainted
modules
Hello, world 3
Module hello-3 loaded, with warnings
```

В ядра версии 2.4 и выше был добавлен механизм контроля лицензий, чтобы иметь возможность предупреждать пользователя об использовании проприетарного (не свободного) кода. Задать условия лицензирования модуля можно с помощью макроопределения `MODULE_LICENSE()`. Ниже приводится выдержка из файла `linux/module.h` (*от переводчика: я взял на себя смелость перевести текст комментариев на русский язык*):

```
/*
 * В настоящее время, для обозначения свободных лицензий, приняты следующие
 * идентификаторы
 *
 *      "GPL"                           [GNU Public License v2 или выше]
 *      "GPL v2"                        [GNU Public License v2]
 *      "GPL and additional rights"     [GNU Public License v2 с дополнительными
правами]
 *      "Dual BSD/GPL"                  [GNU Public License v2
 *                                       или BSD license]
 *      "Dual MPL/GPL"                  [GNU Public License v2
 *                                       или Mozilla license]
 *
 * Кроме того, дополнительно имеются следующие идентификаторы
 *
 *      "Proprietary"                   [проприетарный, не свободный продукт]
 *
 * Здесь присутствуют компоненты, подразумевающие двойное лицензирование,
 * однако, по отношению к Linux они приобретают значение GPL, как наиболее
 * уместное, так что это не является проблемой.
 * Подобно тому, как LGPL связана с GPL
 *
 * На это есть несколько причин
 * 1. modinfo может показать сведения о лицензировании для тех пользователей,
 *    которые желают, чтобы их набор программных компонент был свободным
 * 2. Сообщество может игнорировать отчеты об ошибках (bug reports), относящиеся
 *    к проприетарным модулям
 * 3. Поставщики программных продуктов могут поступать аналогичным образом,
 *    основываясь на своих собственных правилах
 */
```

Точно так же, для описания модуля может использоваться макрос `MODULE_DESCRIPTION()`, для установления авторства -- `MODULE_AUTHOR()`, а для описания типов устройств, поддерживаемых модулем -- `MODULE_SUPPORTED_DEVICE()`.

Все эти макроопределения описаны в файле `linux/module.h`. Они не используются ядром и служат лишь для описания модуля, которое может быть просмотрено с помощью **objdump**. Попробуйте с помощью утилиты **grep** посмотреть, как авторы модулей используют эти макросы (в каталоге `linux/drivers`).

Пример 6. hello-4.c

```
/*
 *  hello-4.c - Демонстрация описания модуля.
 */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#define DRIVER_AUTHOR "Peter Jay Salzman <p@dirac.org>"
#define DRIVER_DESC   "A sample driver"

static int __init init_hello_4(void)
{
        printk(KERN_ALERT "Hello, world 4\n");
        return 0;
}

static void __exit cleanup_hello_4(void)
{
        printk(KERN_ALERT "Goodbye, world 4\n");
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);

/*
 *  Вы можете передавать в макросы строки, как это показано ниже:
 */

/*
 * Запретить вывод предупреждения о "загрязнении" ядра, объявив код под GPL.
 */
MODULE_LICENSE("GPL");

/*
 * или определения:
 */
MODULE_AUTHOR(DRIVER_AUTHOR);    /* Автор модуля */
MODULE_DESCRIPTION(DRIVER_DESC); /* Назначение модуля */

/*
 *  Этот модуль использует устройство /dev/testdevice.  В будущих версиях ядра
 *  макрос MODULE_SUPPORTED_DEVICE может быть использован
 *  для автоматической настройки модуля, но пока
 *  он служит исключительно в описательных целях.
 */
MODULE_SUPPORTED_DEVICE("testdevice");
```

Передача модулю параметров командной строки

Имеется возможность передачи модулю дополнительных параметров командной строки, но делается это не с помощью `argc/argv`.

Для начала вам нужно объявить глобальные переменные, в которые будут записаны входные параметры, а затем вставить макрос `MODULE_PARAM()`, для запуска механизма приема внешних аргументов. Значения параметров могут быть переданы модулю с помощью команд insmod или modprobe. Например: **insmod mymodule.ko myvariable=5**. Для большей ясности, объявления переменных и вызовы макроопределений следует размещать в начале модуля. Пример кода прояснит мое, по общему признанию, довольно неудачное объяснение.

Макрос `MODULE_PARAM()` принимает 2 аргумента: имя переменной и ее тип. Поддерживаются следующие типы переменных

- "b" -- byte (байт);
- "h" -- short int (короткое целое);
- "i" -- integer (целое, как со знаком, так и без знака);
- "l" -- long int (длинное целое, как со знаком, так и без знака);
- "s" -- string (строка, должна объявляться как char*).

Для переменных типа `char *`, **insmod** будет сама выделять необходимую память. Вы всегда должны инициализировать переменные значениями по-умолчанию, не забывайте -- это код ядра, здесь лучше лишний раз перестраховаться. Например:

```
int myint = 3;
char *mystr;

MODULE_PARAM(myint, "i");
MODULE_PARAM(mystr, "s");
```

Параметры-массивы так же допустимы. Целое число, предшествующее символу типа аргумента, обозначает максимальный размер массива. Два числа, разделенные дефисом -- минимальное и максимальное количество значений. Например, массив целых, который должен иметь не менее 2-х и не более 4-х значений, может быть объявлен так:

```
int myintArray[4];
MODULE_PARAM(myintArray, "2-4i");
```

Желательно, чтобы все входные параметры модуля имели значения по-умолчанию, например адреса портов ввода-вывода. Модуль может выполнять проверку переменных на значения по-умолчанию и если такая проверка дает положительный результат, то переходить к автоматическому конфигурированию (вопрос автонастройки будет обсуждаться ниже).

И, наконец, еще одно макроопределение -- `MODULE_PARAM_DESC()`. Оно используется для описания входных аргументов модуля. Принимает два параметра: имя переменной и строку описания, в свободной форме.

Пример 7. hello-5.c

```
/*
 *  hello-5.c - Пример передачи модулю аргументов командной строки.
 */
#include <linux/module.h>
#include <linux/moduleparam.h>
```

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";

/*
 * module_param(foo, int, 0000)
 * Первый параметр -- имя переменной,
 * Второй -- тип,
 * Последний -- биты прав доступа
 * для того, чтобы выставить в sysfs (если ненулевое значение) на более поздней
стадии.
 */

module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

static int __init hello_5_init(void)
{
        printk(KERN_ALERT "Hello, world 5\n=============\n");
        printk(KERN_ALERT "myshort is a short integer: %hd\n", myshort);
        printk(KERN_ALERT "myint is an integer: %d\n", myint);
        printk(KERN_ALERT "mylong is a long integer: %ld\n", mylong);
        printk(KERN_ALERT "mystring is a string: %s\n", mystring);
        return 0;
}

static void __exit hello_5_exit(void)
{
        printk(KERN_ALERT "Goodbye, world 5\n");
}

module_init(hello_5_init);
module_exit(hello_5_exit);
```

Давайте немножко поэкспериментируем с этим модулем:

```
satan# insmod hello-5.o mystring="bebop" myshort=255
myshort is a short integer: 255
myint is an integer: 420
mylong is a long integer: 9999
mystring is a string: bebop

satan# rmmod hello-5
Goodbye, world 5

satan# insmod hello-5.o mystring="supercalifragilisticexpialidocious" myint=100
myshort is a short integer: 1
myint is an integer: 100
mylong is a long integer: 9999
mystring is a string: supercalifragilisticexpialidocious
```

```
satan# rmmod hello-5
Goodbye, world 5

satan# insmod hello-5.o mylong=hello
hello-5.o: `hello' invalid for parameter mylong
```

Модули, состоящие из нескольких файлов

Иногда возникает необходимость разместить исходные тексты модуля в нескольких файлах. В этом случае **kbuild** опять возьмет на себя всю "грязную" работу, а `Makefile` поможет сохранить наши руки чистыми, а голову светлой! Ниже приводится пример модуля, состоящего из двух файлов:

Пример 8. start.c

```
/*
 *  start.c - Пример модуля, исходный текст которого размещен в нескольких
файлах
 */

#include <linux/kernel.h>       /* Все-таки мы пишем код ядра! */
#include <linux/module.h>       /* Необходим для любого модуля ядра */

int init_module(void)
{
        printk("Hello, world - this is the kernel speaking\n");
        return 0;
}
```

Пример 9. stop.c

```
/*
 *  stop.c - Пример модуля, исходный текст которого размещен в нескольких файлах
 */

#include <linux/kernel.h>       /* Все-таки мы пишем код ядра! */
#include <linux/module.h>       /* Необходим для любого модуля ядра */

void cleanup_module()
{
        printk("<1>Short is the life of a kernel module\n");
}
```

Пример 10. Makefile для сборки всех модулей

```
obj-m += hello-1.o
obj-m += hello-2.o
obj-m += hello-3.o
obj-m += hello-4.o
obj-m += hello-5.o
obj-m += startstop.o
startstop-objs := start.o stop.o
```

Задание:

1. Ознакомиться с предложенным материалом.
2. Выполнить приведенные примеры и продемонстрировать преподавателю.