Модуль 2. Модули ядра Linux

Простейший модуль ядра

hello.c:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

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

static void hello_exit(void)
{
      printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Модуль ядра должен иметь как минимум 2 функции: функцию инициализации и функцию выхода. Первая вызывается во время загрузки модуля в пространство ядра, а вторая, соответственно, при выгрузке его. Эти функции задаются с помощью макроопределений: module_init и module_exit.

Знакомство с printk()

NAME
printk - print messages to console log

SYNOPSIS
#include <linux/kernel.h>
int printk(const char*fmt, ...)
DESCRIPTION
Print a formatted message to the kernel console, much like the printf function of the stdio library. Normally, the message is written to the physical console device of the computer, although this behavior can be changed with the register_console function. Messages are also stored in a message log book.
The generated string may also start with a message priority code, which sets the priority of the message. The priority code strings are of the form <n> where n is a number from 0 - 7. The following macros are defined in the <linux/kernel.h> header file:

KERN_EMERG
System is unuseable
KERN_ALERT
Action must be taken immediately
KERN_CRIT
Critical conditions
KERN_ERR
Error conditions
KERN_WARNING
Warning conditions
KERN_NOTICE
Normal but significant condition
KERN_INFO
Informational
KERN_DEBUG
Debug-level messages
For example

```
printk(KERN_NOTICE "Hello, world.\n");
```

does the expected thing.
RETURN VALUE
Returns the number of characters written to the log.
AVAILABILITY
Linux 1.0+
SEE ALSO

```
register_console(9), syslog(2)
kernel/printk.c
AUTHOR
Stephen Williams (steve@icarus.com)
BUGS
float and double formats are not supported. Floats and
doubles do not belong inside the kernel anyhow.
The printk implementation protects itself from
interruption, so in principle it can be used in
interrupts handlers and critical sections. However, there
are no guarantees about the console function that is
registered.
```

Для вывода информационных и отладочных сообщений используется функция «printk()»:

```
int printk(const char * fmt, ...);
```

Основное назначение этой функции — реализация механизма регистрации событий и предупреждений. Иными словами эта функция для записи в лог ядра некой информации. 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). Первому параметру может предшествовать (а может и не предшествовать, но тогда это просто означает, что используется значение по умолчанию) константа квалификатор, определяющая уровень сообщений. Эта предшествующая константа в вызове printk() не является отдельным параметром (не отделяется запятой), и (как видно из примеров) представляет собой символьную строку определённого вида, которая конкатенируется с первым параметром (являющимся, в общем случае, форматной строкой). Если данная константа отсутствует, то для этого сообщения устанавливается уровень вывода по умолчанию (часто это KERN_WARNING).

Существует всего 8 кодов приоритета:

```
KERN_EMERG   "<0>" /* system is unusable               */
KERN_ALERT   "<1>" /* action must be taken immediately */
KERN_CRIT    "<2>" /* critical conditions              */
KERN_ERR     "<3>" /* error conditions                 */
KERN_WARNING "<4>" /* warning conditions               */
KERN_NOTICE  "<5>" /* normal but significant condition */
KERN_INFO    "<6>" /* informational                    */
```

```
KERN_DEBUG   "<7>" /* debug-level messages                  */
```

Данные макроопределения находятся в следующх файлах:

>= 3.6	<linux/kern_levels.h>
>= 2.6.37	<linux/printk.h>
…	<linux/kernel.h>

Макроопределения __init и __exit

Макроопределение __init вынуждает ядро освободить память, занимаемую функцией, после выполнения инициализации модуля, правда относится это только к встроенным модулям и не имеет никакого эффекта для загружаемых модулей. 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.

Оба этих макроса определены в файле <linux/init.h> и отвечают за освобождение неиспользуемой памяти в ядре.

Сборка модулей ядра

Процесс сборки модулей существенно отличается от используемого для приложений пользовательского пространства; ядро - это большая автономная программа с подробными и точными требованиями о том, как кусочки собирать вместе. Система сборки ядра довольно сложна и мы рассмотрим только небольшую её часть. Файлы, находящиеся в директории Documentation/kbuild исходных кодов ядра, являются обязательными для чтения любым желающим понять всё, что на самом деле происходит под поверхностью. Есть некоторые предварительные условия, которые необходимо выполнить, прежде чем вы сможете собрать модули ядра. Во- первых, убедитесь, что вы имеете достаточно свежие версии компилятора, утилит модулей и других необходимых утилит. Файл Documentation/changes в каталоге документации ядра всегда содержит список необходимых версий утилит; вы должны проверить это перед тем, как двигаться дальше. Попытка построить ядро (и его модули) с неправильными версиями утилит может привести к бесконечным тонким, сложным проблемам. Заметим, что иногда слишком новая версия компилятора может быть столь же проблематичной, как и слишком старая; исходный код ядра делает много предположений о компиляторе и новые релизы могут иногда сломать что-то на некоторое время.

После того как вы всё установили, создать Makefile для модуля просто. Фактически, для примера "hello world", приведённого ранее в этой главе, достаточно будет одной строчки:

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

Как же работает этот Makefile? В конце концов, приведённая выше строчка выглядит не как обычный Makefile. Ответ конечно же в том, что всё остальное делает система сборки ядра. Определение выше (которое использует расширенный синтаксис, предоставляемый GNU make) заявляет, что требуется собрать один модуль из объектного файла hello.o. Результирующий модуль после сборки из объектного файла именуется как hello.ko.

В системе имеется каталог «/lib/modules/[версия ядра]/», в нем находятся модули для соответствующей версии ядра. Также там имеется build – это символическая ссылка на заголовки библиотек ядра (kernel-headers), которые находятся в каталоге «/usr/src/linux-headers-[версия ядра]/». Если у вас еще нет kernel-headers, то их нужно скачать («sudo apt-get install linux-headers-[версия ядра]»). Чтобы узнать версию текущего ядра, работающей в вашей системе, можно выполнить команду «uname –r». Вопросы несовпадения версий работающего ядра и версии ядра, под которую был собран модуль рассмотрим ниже.

Команда сборки модуля будет выглядеть следующим образом:

```
make -C /lib/modules/[версия ядра]/build M=`pwd` modules
```

Эта команда начинается со смены своего каталога на указанный опцией -C (то есть на ваш каталог исходных кодов ядра). Там она находит Makefile верхнего уровня ядра. Опция M= заставляет Makefile вернуться обратно в директорию исходников вашего модуля, прежде чем попытаться построить целевой модуль. Эта задача, в свою очередь, ссылается на список модулей, содержащихся в переменной obj-m, который мы определили в наших примерах как module.o. Ввод предыдущей команды make может стать через некоторое время утомительным, так что разработчики ядра разработали своего рода идиому Makefile, которая делает жизнь легче для модулей, собираемых вне дерева ядра. Весь фокус в том, чтобы написать Makefile следующим образом:

```
ifeq ($(KERNELRELEASE),)

   KERNELDIR ?= /lib/modules/$(shell uname -r)/build
   PWD := $(shell pwd)

modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules_install:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install

clean:
    rm -rf *.o *~ core .depend .*.cmd *.ko
*.mod.c .tmp_versions Module.symvers modules.order

.PHONY: modules modules_install clean

else
     # called from kernel build system: just declare what
our modules are
    obj-m := hello.o
endif
```

И снова мы видим расширенный синтаксис GNU make в действии. Этот Makefile читается при типичной сборке дважды. Когда Makefile вызывается из командной строки, он замечает, что переменная KERNELRELEASE не установлена. Он определяет расположение каталога исходных текстов ядра, воспользовавшись тем, что символическая ссылка build в директории установленных модулей указывает обратно на дерево сборки ядра. Если у вас на самом деле не

работает ядро, для которого вы делаете сборку, можно добавить параметр KERNELDIR= в командную строчку, установив переменную KERNELDIR среды окружения или переписать строчку, которая задаёт KERNELDIR в Makefile. После того, как дерево исходных текстов ядра было найдено, Makefile запускает цель default:, которая запускает вторую команду make (параметризованную как $(MAKE) в Makefile), чтобы вызвать систему сборки ядра, как описано выше. При втором чтении Makefile устанавливает obj-m и о реальном создании модуля заботятся Makefile-ы ядра.

Вопросы документирования модулей

В ядре, начиная с версии 2.4, 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(). 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.

Различные декларации MODULE_ могут появиться в любом месте вашего исходного файла вне функции. Однако, по относительно недавнему соглашению в коде ядра эти декларации размещаются в конце файла.

Передача модулю параметров командной строки

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

Макрос module_param() принимает 3 аргумента: имя переменной, ее тип и права доступа. Если 3 переменая отсутствует или равна 0, данный параметр не будет представлен в sysfs. 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.

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.

Для сборки необходимо указать в Makefile следующие строки:

```
obj-m := hello.o
startstop-objs := start.o stop.o
```

Сборка модулей под существующее ядро

Если вы лишь установили дерево с исходными текстами ядра и использовали их для сборки своего модуля, то в большинстве случаев при попытке загрузить его в работающее ядро, вы получите следующее сообщение об ошибке:

```
insmod: error inserting 'your_module_name.ko': -1 Invalid module format
```

Ядро отказывается "принимать" модуль из-за несоответствия версий (точнее — из-за несоответствия сигнатур версий). Сигнатура версии сохраняется в объектном файле в виде статической строки, начинающейся со слова vermagic:. Эта строка вставляется во время компоновки модуля с файлом init/vermagic.o. Просмотреть сигнатуру версии (так же как и некоторые дополнительные сведения) можно посредством команды:

```
modinfo module.ko
```

Но было бы ещё хуже, если бы этот модуль, собранный не для данного ядра, удалось загрузить, так как малейшая ошибка в коде модуля ядра, в отличие от процессов пространства пользователя, может привести к немедленной смерти всей операционной системы. Иногда ядро даже не успевает вывести «предсмертное» сообщение Oops, которым обычно диагностируются критические ошибки (аналог «синего экрана смерти» в MS Windows). Вот к чему могут привести ошибки в коде модуля ядра.