Модуль 7 . Блочные устройства

Блочные устройства

Блочные устройства в UNIX и Linux — это устройства хранения с произвольным доступом, над которыми размещаются файловые системы. Отсюда и вытекают отличия блочных устройств от символьных устройств, которые были рассмотрены ранее. Эти различия в основном связаны с двумя факторами:

1. Для блочных устройств первостепенным фактором является производительность (т.е. скорость). Для символьных устройств производительность может быть не так важна, так как многие символьные устройства могут работать ниже своей максимальной скорости, и производительность системы в целом от этого не пострадает. Но для блочных устройств скорость — это одно из конкурентных преимуществ операционной системы на рынке. Поэтому основная часть усилий при разработке блочного модуля сосредотачивается на обеспечении требуемой производительности.
2. Символьные устройства получают запросы ввода-вывода из процессов пространства пользователя, выполняющих операции read() или write(). Блочные устройства могут получать запросы ввода-вывода как из пользовательских процессов, так и из кода ядра (модулей ядра), например, при монтировании дисковых устройств. Поэтому запросы ввода-вывода должны сначала

попасть в блочную подсистему ввода-вывода ядра, и только затем быть переданными ею непосредственно драйверу , осуществляющему обмен с устройством.

Блочное устройство обеспечивает обмен блоками данных. Блок — это единица данных фиксированного размера. Размер блока определяется ядром, но чаще всего он совпадает с размером страницы аппаратной архитектуры, и для 32-битной архитектуры x86 составляет 4096 байт. Оборудование хранит данные на физических носителях, разбитых на сектора. Исторически сложилось так, что последние несколько десятилетий аппаратное обеспечение создавалось для работы с секторами размером 512 байт. В последние годы в новых устройствах появилась тенденция оперировать с большими секторами (4096 байт).

В принципе, модуль блочного устройства может "наследовать" технику символьных устройств, детально рассмотренных ранее, и для некоторых операций это так и есть. Но сами операции, осуществляющие поддержку обмена данными, строятся по-другому. Мы начнём с рассмотрения аналогий между символьными и блочными устройствами, а затем перейдем уже к специфическим особенностям блочных устройств.

Регистрация устройства

Регистрация блочного устройства во многом совпадает с регистрацией символьного устройства, но с коррекцией на специфику блочных устройств, так, регистрация устройства и его последующее

удаление инициируются вызовами API:

```
int register_blkdev( unsigned major, const char* name );
void unregister_blkdev( unsigned major, const char* name );
```

Аналогично символьным устройствам, в качестве параметра major в вызов register_blkdev() может быть передано:

принудительное значение, которое следует присвоить в качестве старшего номера для устройств такого класса;

нулевое значение, что позволит системе присвоить major первое свободное значение.

В любом случае вызов register_blkdev() возвратит текущее значение major или отрицательное значение, сигнализирующее об ошибке. Обычно это происходит, когда для major задаётся принудительное значение, уже занятое в системе.

В качестве name в этот вызов передаётся родовое имя класса устройств, например, для дисков xda, xdb, ... , создаваемых в примере ниже, это будет "xd". Регистрация имени устройства создаёт соответствующую запись в файле /proc/devices, но не создаёт самого устройства в /dev:

```
$ cat /proc/devices | grep xd
252 xd
```

Начиная с ядра 2.6 и старше, в принципе, регистрацию с помощью register_blkdev() можно и не проводить, но обычно это делается, как дань традиции.

Сразу перед вызовом register_blkdev() или после него для каждого устройства (привода) драйвера выполняют инициализацию очереди обслуживания, связанной с устройством, как показано ниже.

```
spinlock_t xda_lock;
struct request_queue* xda_request_queue;
...
spin_lock_init( &xda_lock );
if( !( xda_request_queue = blk_init_queue( &xda_request_func,
&xda_lock ) ) ) {
```

Показанный фрагмент создаёт очередь обслуживания запросов xda_request_queue в ядре и увязывает её с примитивом синхронизации xda_lock, который будет использоваться для монополизации выполняемых операций с этой очередью. Также в нём для очереди xda_request_queue определяется функция обработки запросов, которая будет вызываться ядром при каждом требовании на обработку запроса чтения или записи. Эта функция имеет следующий прототип:

```
static void xda_request_func( struct request_queue *q ) { ... }
```

Отметим, что обработка запросов с помощью очереди ядра не является единственным способом обеспечения операций чтения-записи. Но такой способ используют 95% драйверов. Остающиеся 5% реализуют прямое выполнение запросов по мере их поступления, и такой способ также будет рассмотрен далее.

Но все эти подготовительные операции не приблизили нас к созданию отображения блочного устройства в каталог /dev. Для реального создания отображения каждого диска в /dev необходимо:

1. создать для этого устройства структуру struct gendisk, описанную в файле linux/genhd.h и являющуюся описанием каждого диска в ядре;
2. заполнить эту структуру и зарегистрировать её в ядре.

Примечание: Экземпляром точно такой же структуры в ядре

будет описываться каждый раздел (partition) диска, создаваемый с помощью утилит fdisk или parted. Но разработчику не нужно беспокоится об этих экземплярах, так как их автоматически создадут утилиты, производящие разбивку диска.

Так, структура struct gendisk представляет собой динамически создаваемую структуру, связанную с ядром и требующую специальных манипуляций со стороны ядра для инициализации. Поэтому процесс её создания отличается от аналогичного процесса для других структур. Для создания этой структуры и её последующего уничтожения после использования существуют вызовы:

```
struct gendisk* alloc_disk( int minors );
void del_gendisk( struct gendisk* );
```

Параметр minors в вызове alloc_disk() указывает числом младших номеров, резервируемых для отображения этого диска и всех его разделов в каталоге /dev.

Примечание : Как показывает практика , динамическое изменение поля minors в существующей структуре struct gendisk, не приведет к желаемому результату. Это связано с тем, что процесс инициализации очень сложен, и в зависимости от значения minors системой выделяется соответствующее число слотов (дочерних структур struct gendisk), т.е. изменение значения поля minors "постфактум" смысла не имеет и может даже оказаться вредным.

Диски с разметкой MBR и GPT

Параметр minors в вызове alloc_disk() заслуживает отдельного рассмотрения, так как если задать для minors значение 4, то с помощью

fdisk вы сможете создать до 4-х первичных (primary) разделов в MBR этого диска (/dev/xda1, …, /dev/xda4). Но вы не сможете в этом случае создать ни единого расширенного (extended) раздела, потому, что первый же созданный логический (logical) раздел внутри расширенного получит номер 5:

```
# fdisk -l /dev/xda
...
Устр-во Загр     Начало      Конец      Блоки   Id  Система
/dev/xda1            1       4000       2000   83  Linux
/dev/xda2         4001       8191       2095+   5  Расширенный
/dev/xda5         4002       8191       2095   83  Linux
# ls -l /dev/xda*
brw-rw---- 1 root disk 252, 0 нояб. 12 10:44 /dev/xda
brw-rw---- 1 root disk 252, 1 нояб. 12 10:44 /dev/xda1
brw-rw---- 1 root disk 252, 2 нояб. 12 10:44 /dev/xda2
brw-rw---- 1 root disk 252, 5 нояб. 12 10:46 /dev/xda5
```

Примечание: Пользователи часто интересуются, почему никакими сторонними средствами при разбивке диска не удаётся создать больше 16 разделов. Хотя fdisk не умеет этого делать, но, по определению, цепочка вложенных extended разделов может быть безграничной, и сторонние средства разбивки позволяют создавать такие вложенные структуры расширенных разделов. На самом деле, в принципе создать большее число разделов на физическом диске можно, но драйвер, поддерживающий этот диск, не способен отобразить эти созданные разделы.

Всё сказанное выше относится к способу разбивки диска в стандарте разметки MBR (Master Boot Record), который используется уже на протяжении последних 35 лет. Но в настоящее время происходит давно назревший переход к разметке диска в стандарте GPT (GUID

Partition Table). Основные отличительные стороны GPT в контексте рассмотрения модулей блочных устройств заключаются в том, что:

- используются только первичные разделы;
- число таких разделов может доходить до 128;
- полностью изменены идентификаторы типов разделов и они теперь 32-битные (например, для Linux файловых систем — 8300 вместо прежних 83).

Так как утилита fdisk не поддерживает диски, отформатированные в GPT, то для работы с ними используются утилиты parted (gparted) или gdisk:

```
$ sudo parted /dev/sdb
GNU Parted 3.0
Используется /dev/sdb
(parted) print
Модель: Ut163 USB2FlashStorage (scsi)
Диск /dev/sdb: 1011MB
Размер сектора (логич./физич.): 512B/512B
Таблица разделов: gpt
Disk Flags:.

Номер Начало  Конец   Размер  Файловая система Имя          Флаги
1     1049kB  53,5MB  52,4MB  ext2             EFI System    загрузочный,
legacy_boot
10    53,5MB   578MB   524MB                   Microsoft basic data
20    578MB   1011MB   433MB                   Linux filesystem
```

Утилитами для работы с GPT-дисками можно создать до 128 разделов (или любое число дисков с номерами раздела 1...128). Ниже показаны 18 разделов на диске, созданных с помощью gdisk:

```
$ sudo gdisk -l /dev/sdf
GPT fdisk (gdisk) version 0.8.4
…
```

Из показанного очевидно, что при переходе на новый стандарт форматирования ожидаются серьезные изменения, которые следует

учитывать при создании модулей блочных устройств. Например, в качестве значения параметра minors в вызове alloc_disk() должно указываться 128.

Заполнение структуры

Мы остановились на создании структуры struct gendisk, и теперь можно вернуться к заполнению её полей. Так как эта структура весьма объёмна, то отметим только те поля, которые нужно заполнить под свой драйвер (есть ещё несколько полей, упомянутых в следующем примере, которые требуют "технического" заполнения):

```
struct gendisk {
        int major;
        int first_minor;
        int minors;
        char disk_name[DISK_NAME_LEN];
...
        const struct block_device_operations *fops;
...
}
```

Перечислим важнейшие поля:

major— старший номер устройства, который присваивается устройству принудительно или возвращается в результате вызова register_blkdev(0, ...);

minors— максимальное число разделов, обслуживаемых диском; это поле заполняется вызовом alloc_disk() и впоследствии не должно меняться;

first_minors— номер minor, представляющий сам диск в /dev (например, dev/xda), последующие разделы диска (в пределах их числа minors) получат соответствующие младшие номера, например, для /dev/

xda5 будет использован номер first_minor+5:

```
# ls -l /dev/xda*
brw-rw---- 1 root disk 252, 0 нояб. 12 10:44 /dev/xda
brw-rw---- 1 root disk 252, 1 нояб. 12 10:44 /dev/xda1
brw-rw---- 1 root disk 252, 2 нояб. 12 10:44 /dev/xda2
brw-rw---- 1 root disk 252, 5 нояб. 12 10:46 /dev/xda5
```

disk_name — имя диска, с которым он будет отображаться в /dev или родовое имя устройств, так как драйвер может обслуживать более одного привода устройства; указывается в вызове типа:

```
register_blkdev( major, MY_DEVICE_NAME );
```

Теперь мы можем персонифицировать имя для конкретного привода, как показано ниже:

```
snprintf( disk_name, DISK_NAME_LEN - 1, "xd%c",  'a' + i );
```

В дальнейшем эти имена появятся в /proc:

```
$ cat /proc/partitions
major minor  #blocks  name

   8        0   58615704 sda
   8        1   45056000 sda1
   8        2    3072000 sda2
   8        3   10485760 sda3
...
 252        0       4096 xda
 252        1       2000 xda1
 252        2          1 xda2
 252        5       2095 xda5
 252       16       4096 xdb
...
 252       48       4096 xdd
```

fops — адрес таблицы операций устройства, которую мы детально обсудим в следующих статьях.

Кроме заполнения полей структуры struct gendisk примерно в этом месте выполняется установка полной ёмкости устройства, выраженной в секторах по 512 байт:

```
inline void set_capacity( struct gendisk*, sector_t size );
```

Завершение регистрации

Теперь структура struct gendisk выделена и заполнена, но диск ещё не готов к использованию. Чтобы завершить подготовительные операции, следует вызвать метод add_disk(struct gendisk* gd), передав в качестве параметра подготовленную структуру gd.

Вызов add_disk() достаточно «опасный», так как, как только он произойдёт, диск становится активным и его методы могут быть вызваны в любое время. На самом же деле, первые вызовы происходят ещё до того, как произойдёт возврат из самого вызова add_disk() (это можно увидеть в системном журнале). Эти вызовы связаны с попытками ядра вычитать начальные сектора диска в поисках таблицы разделов (MBR или GDT). Если к моменту вызова add_disk() драйвер ещё не полностью или некорректно инициализирован, то, с большой вероятностью, вызов приведёт к ошибке драйвера и, через некоторое время, к полному краху системы.

Операции, поддерживаемые блочными устройствами

В предыдущих пунктах мы заполнили в структуре описания диска struct gendisk поле fops— таблицу операций блочного устройства. Таблица функций struct block_device_operations выполняет для блочного устройства ту же роль, что и таблица файловых операций для символьного устройства struct file_operations, рассматривавшаяся ранее. Ниже приведен сокращенный фрагмент таблицы block_device_operations из ядра версии 3.5.

```
struct block_device_operations {
        int (*open) (struct block_device *, fmode_t);
        int (*release) (struct gendisk *, fmode_t);
        int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        unsigned int (*check_events) (struct gendisk *disk, unsigned int
clearing);
        void (*unlock_native_capacity) (struct gendisk *);
        int (*revalidate_disk) (struct gendisk *);
        int (*getgeo)(struct block_device *, struct hd_geometry *);
        struct module *owner;
...
};
```

В данном фрагменте всё должно быть понятно, так как его содержание во многом напоминает аналогичную таблицу для символьных устройств. Операции open() и release()— это функции, вызываемые при открытии и закрытии устройства, но довольно часто они вызываются неявно, поэтому мы не будем описывать их.

Листинг 1. Общие определения

```
static struct block_device_operations mybdrv_fops = {
  .owner = THIS_MODULE,
  .ioctl = my_ioctl,
  .getgeo = my_getgeo
};
```

Листинг 2. Пример реализации операций getgeo() и ioctl()

```
static int my_getgeo( struct block_device *bdev, struct hd_geometry *geo ) {
  unsigned long sectors = ( diskmb * 1024 ) * 2;
  DBG( KERN_INFO "getgeo\n" );
  geo->heads = 4;
  geo->sectors = 16;
  geo->cylinders = sectors / geo->heads / geo->sectors;
  geo->start = geo->sectors;
  return 0;
};
static int my_ioctl( struct block_device *bdev, fmode_t mode,
                unsigned int cmd, unsigned long arg ) {
  DBG( "ioctl cmd=%d\n", cmd );
  switch( cmd ) {
    case HDIO_GETGEO: {
      struct hd_geometry geo;
      LOG( "ioctk HDIO_GETGEO\n" );
      my_getgeo( bdev, &geo );
```

```
        if( copy_to_user( (void __user *)arg, &geo, sizeof( geo ) ) )
          return -EFAULT;
        return 0;
      }
      default:
        DBG( "ioctl unknown command\n" );
        return -ENOTTY;
    }
}
```

Примечание: Используемые макросы ERR(), LOG(), DBG() определены только для компактности кода и скрывают за собой вызов printk(). Их исходный код можно найти в листинге 3.

Современное ядро никак не связано с конфигурацией блочного устройства, которое рассматривается как линейный массив секторов (в наших примерах это массив секторов в памяти). Также для работы системы с блочным устройством не имеет значения, как оно разбито в терминах цилиндров, головок и числа секторов на дорожку. Но многие утилиты GNU для работы с дисковыми устройствами (fdisk, hdparm, ...) предполагают получение информации о геометрии диска. Поэтому, чтобы не создавать препятствий для работы этих утилит, целесообразно реализовать показанные выше операции, так как без них может оказаться невозможным создание разделов на устройстве. Проверим работу функции getgeo() из листинга 2.

```
# insmod block_mod_s.ko# hdparm -g /dev/xdd
/dev/xdd:
 geometry      = 128/4/16, sectors = 8192, start = 16
```

В таблице операций struct block_device_operations находит отражение тот факт, что эти операции предназначены именно для блочного устройства. Например, при разработке устройства со сменными носителями для проверки несменяемости носителя при

каждом новом открытии устройства в теле функции open() следует вызывать функцию ядрacheck_disk_change(struct block_device*).

В свою очередь для проверки фактической смены носителя функция check_disk_change() выполняет вызов метода из таблицы операций:

```
int (*media_changed)( struct gendisk* );
```

Если носитель сменился, то функция media_changed() возвращает ненулевое значение, и в этом случае ядро вызовет ещё один метод из таблицы операций:

```
int (*revalidate_disk)( struct gendisk *);
```

Этот метод, реализуемый программистом, должен выполнить операции, необходимые для подготовки драйвера для работы с новым носителем (если добавляемым устройством является RAM-диск, то вызов должен обнулить область памяти устройства). После осуществления вызова revalidate_disk(), ядро тут же предпримет попытку заново считать таблицу разделов устройства. Именно так обрабатываются сменные носители в блочных устройствах.

Примечание: Нужно различать устройство со сменным носителем и сменное устройство. К устройствам со сменным носителем относятся DVD/CD-дисководы и подобные устройства. А сменные устройства — это внешние жёсткие диски, USB флеш-диски и т.д. Реинициализация драйвера в этих случаях происходит совершенно по разному.

И последние замечания относительно таблицы операций:

При изменении версии ядра в struct block_device_operations

также происходят изменения, например, вызов swap_slot_free_notify() появился в таблице только начиная с ядра 2.6.35 как указывается в данной статье;

Также существующие и описанные в публикациях методы могут быть объявлены устаревшими (deprecated, т.е. могут быть удалены в будущих версиях), как, например, метод media_changed() (с версии 2.6.38), вместо которого предложен новый метод check_events():

```
unsigned int (*check_events) (struct gendisk *disk, unsigned int clearing);
  /* ->media_changed() is DEPRECATED, use ->check_events() instead */
 int (*media_changed) (struct gendisk *);
```

Обработка запросов к блочному устройству

К данному моменту мы рассмотрели многие аспекты функционирования блочных устройств за исключением самих операции чтения и записи данных. Так вот сами операции чтения и записи в драйвере блочного устройства отсутствуют! Они выполняются по-другому:

- получив запрос на чтение или запись ядро помещает его в очередь, связанную с драйвером (которую мы инициализировали ранее);
- запросы в очереди ядро сортирует так, чтобы оптимизировать выполнение последовательности запросов (например , минимизировать перемещения головок жёсткого диска);
- очередной (текущий крайний) запрос из очереди передаётся на обработку функции драйвера, которую мы определили для обработки запросов;
- направление (ввод или вывод) и параметры запроса (начальный

сектор, число секторов и др.) указываются в самом теле запроса.

Именно так, хотя и очень приблизительно, можно описать схему работы драйвера блочного устройства.

К этому времени мы рассмотрели всю информацию, необходимую для создания работающего примера модуля блочного устройства. Рассмотрим пример реализации.

Листинг 3. Включаемый файл для модуля блочного устройства (файл common.h)

```
#include <linux/module.h>
#include <linux/vmalloc.h>
#include <linux/blkdev.h>
#include <linux/genhd.h>
#include <linux/errno.h>
#include <linux/hdreg.h>
#include <linux/version.h>

#define KERNEL_SECTOR_SIZE    512

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,35)
#define blk_fs_request(rq)      ((rq)->cmd_type == REQ_TYPE_FS)
#endif

static int diskmb = 4;
module_param_named( size, diskmb, int, 0 ); // размер диска в Mb, по
умолчанию - 4Mb
static int debug = 0;
module_param( debug, int, 0 );              // уровень отладочных сообщений

#define ERR(...) printk( KERN_ERR "! "__VA_ARGS__ )
#define LOG(...) printk( KERN_INFO "+ "__VA_ARGS__ )
#define DBG(...) if( debug > 0 ) printk( KERN_DEBUG "#
"__VA_ARGS__ )
```

Код модуля, представленный с сокращениями в листинге 4, предполагает реализацию трёх альтернативных стратегий реализации обмена данными. Выбор осуществляется при запуске модуля путём установки значения параметра mode, которое определяет, какая функция обработки будет использоваться. Поэтому (для облегчения

понимания), сначала мы рассмотрим полный код модуля, но исключим из него реализации самих обработчиков обмена, а в следующей статье подробно изучим как реализуются обработчики для каждой стратегии.

Листинг 4. Модуль блочного устройства (файл block_mod_s.c)

```
#include "../common.h"

static int major = 0;
module_param( major, int, 0 );
static int hardsect_size = KERNEL_SECTOR_SIZE;
module_param( hardsect_size, int, 0 );
static int ndevices = 4;
module_param( ndevices, int, 0 );
// различные стратегии обмена данными
enum {     RM_SIMPLE  = 0,     // простой запрос с обработкой очереди
    RM_FULL    = 1,       // запрос с обработкой вектора BIO
    RM_NOQUEUE = 2, // прямое выполнение запроса без очереди
    };
static int mode = RM_SIMPLE;
module_param( mode, int, 0 );

static int nsectors;

struct disk_dev {              // внутренняя структура нашего устройства
  int size;                  // размер устройства в секторах
  u8 *data;                    // массив с данными
  spinlock_t lock;             // флаг блокировки устройства
  struct request_queue *queue; // очередь запросов к устройству
  struct gendisk *gd;
};

static struct disk_dev *Devices = NULL;
...
// простой запрос с обработкой очереди
static void simple_request( struct request_queue *q ) {
...
}

// запрос с обработкой вектора BIO
static void full_request( struct request_queue *q ) {
...
}

// прямое выполнение запроса без очереди
static void make_request( struct request_queue *q, struct bio *bio ) {
```

```
...
}

#include "../ioctl.c"
#include "../common.c"

#define MY_DEVICE_NAME "xd"
#define DEV_MINORS   16

// настройка внутреннего устройства
static void setup_device( struct disk_dev *dev, int which ) {
  memset( dev, 0, sizeof( struct disk_dev ) );
  dev->size = diskmb * 1024 * 1024;
  dev->data = vmalloc( dev->size );
  if( dev->data == NULL ) {
    ERR( "vmalloc failure.\n" );
    return;
  }
  spin_lock_init( &dev->lock );
  switch( mode ) { // выбор режима выполнения операции
    case RM_NOQUEUE:
      dev->queue = blk_alloc_queue( GFP_KERNEL );
      if( dev->queue == NULL ) goto out_vfree;
      blk_queue_make_request( dev->queue, make_request );
      break;
    case RM_FULL:
      dev->queue = blk_init_queue( full_request, &dev->lock );
      if( dev->queue == NULL ) goto out_vfree;
      break;
    default:
      LOG( "bad request mode %d, using simple\n", mode );
    case RM_SIMPLE:
      dev->queue = blk_init_queue( simple_request, &dev->lock );
      if( dev->queue == NULL ) goto out_vfree;
      break;
  }
  // установка раздела аппаратного сектора
  blk_queue_logical_block_size( dev->queue, hardsect_size );
  dev->queue->queuedata = dev;
  dev->gd = alloc_disk( DEV_MINORS );
  if( ! dev->gd ) {
    ERR( "alloc_disk failure\n" );
    goto out_vfree;
  }
  dev->gd->major = major;
  dev->gd->minors = DEV_MINORS;
  dev->gd->first_minor = which * DEV_MINORS;
```

```
  dev->gd->fops = &mybdrv_fops;
  dev->gd->queue = dev->queue;
  dev->gd->private_data = dev;
  snprintf( dev->gd->disk_name, 32, MY_DEVICE_NAME"%c", which +
'a' );
  set_capacity( dev->gd, nsectors * ( hardsect_size /
KERNEL_SECTOR_SIZE ) );
  add_disk( dev->gd );
  return;
out_vfree:
  if( dev->data ) vfree( dev->data );
}

static int __init blk_init( void ) {
  int i;
  nsectors = diskmb * 1024 * 1024 / hardsect_size;
  major = register_blkdev( major, MY_DEVICE_NAME );
  if( major <= 0 ) {
    ERR( "unable to get major number\n" );
    return -EBUSY;
  }
  // выделение массива для хранения данных устройства
  Devices = kmalloc( ndevices * sizeof( struct disk_dev ), GFP_KERNEL );
  if( Devices == NULL ) goto out_unregister;
  for( i = 0; i < ndevices; i++ ) // инициализировать каждое устройство
    setup_device( Devices + i, i );
  return 0;
out_unregister:
  unregister_blkdev( major, MY_DEVICE_NAME );
  return -ENOMEM;
}


static void blk_exit( void ) {
  int i;
  for( i = 0; i < ndevices; i++ ) {
    struct disk_dev *dev = Devices + i;
    if( dev->gd ) {
      del_gendisk( dev->gd );
      put_disk(dev->gd);
    }
    if( dev->queue ) {
      if( mode == RM_NOQUEUE )
        blk_put_queue( dev->queue );
      else
        blk_cleanup_queue( dev->queue );
    }
```

```
    if( dev->data ) vfree( dev->data );
  }
  unregister_blkdev( major, MY_DEVICE_NAME );
  kfree( Devices );
}
MODULE_AUTHOR( "Jonathan Corbet" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_LICENSE( "GPL v2" );
MODULE_VERSION( "1.5" );
```

Детали реализации

Использование очереди для обработки запросов ввода-вывода

В первом способе обработки запросов (выбираемом, когда значение параметра mode равно 0) используется классический подход к написанию модулей блочных устройств (большая часть драйверов использует эту стратегию). Обработка осуществляется двумя функциями, приведенными в листинге 1.

Листинг 1. Обработка запросов с помощью очереди

```
//функция, непосредственно выполняющая обмен данными
static int transfer( struct disk_dev *dev, unsigned long sector,
                     unsigned long nsect, char *buffer, int write ) {
  unsigned long offset = sector * KERNEL_SECTOR_SIZE;
  unsigned long nbytes = nsect * KERNEL_SECTOR_SIZE;
  if( (offset + nbytes) > dev->size ) {
    ERR( "beyond-end write (%ld %ld)\n", offset, nbytes );
    return -EIO;
  }
  if( write )
    memcpy( dev->data + offset, buffer, nbytes );
  else
     memcpy( buffer, dev->data + offset, nbytes );
  return 0;
}
//функция, организующая обработку запросов с использованием очереди
static void simple_request( struct request_queue *q ) {
  struct request *req;
  unsigned nr_sectors, sector;
  DBG( "entering simple request routine\n" );
  req = blk_fetch_request( q );
  while( req ) {
```

```
    int ret = 0;
    struct disk_dev *dev = req->rq_disk->private_data;
    if( !blk_fs_request( req ) ) {
      ERR( "skip non-fs request\n" );
      __blk_end_request_all( req, -EIO );
      req = blk_fetch_request( q );
      continue;
    }
    nr_sectors = blk_rq_cur_sectors( req );
    sector = blk_rq_pos( req );
    ret = transfer( dev, sector, nr_sectors, req->buffer, rq_data_dir( req ) );
    if( !__blk_end_request_cur( req, ret ) )
      req = blk_fetch_request( q );
  }
}
```

Ещё не обработанные или не завершённые запросы выбираются из очереди ядра request_queue вызовом:

```
struct request* req blk_fetch_request( struct request_queue* );
```

Запрос может потребовать выполнения специальных (управляющих) операций, не относящихся к чтению или записи и переданными «не тому» устройству, но такие запросы можно отфильтровать вызовом:

```
bool blk_fs_request( struct request* );
```

Если запрос признан корректным, то из него извлекаются следующие параметры:

- blk_rq_pos(struct request*)— начальный сектор для обмена;
- blk_rq_cur_sectors(struct request*)— число секторов подлежащих обмену;
- (struct request*)->buffer— буфер в пространстве ядра;
- rq_data_dir(struct request*)— направление обмена (запись или чтение).

Далее управление передаётся функции transfer(). Эта функция

проверяет, что запрашиваемые данные не находятся за пределами пространства устройства (это была бы грубая ошибка), и осуществляет копирование данных в или из буфера ядра . Дальнейшее распространение полученных данных к получателю запроса выполняет ядро.

Независимо от того, как был выполнен (удачно или нет) запрос, ядру следует сообщить о его завершении и переходе к следующему запросу из очереди. Для уведомления ядра используются вызовы:

```
__blk_end_request_all( struct request*, int errno );
__blk_end_request_cur( struct request*, int errno );
```

В параметре errno передаётся результат операции: 0 в случае успеха, отрицательный код (как принято в ядре) в случае ошибки. После этих вызовов текущий запрос считается обработанным и удаляется из очереди.

Примечание: В API блочных операций практически для всех вызовов «с подчёркиванием» существуют парные вызовы «без подчёркивания», например, __blk_end_request_all() соответствует blk_end_request_all(). Разница между ними состоит в том, что методы с подчёркиванием сами выполняют блокировку для монопольного использования очереди (мы инициализировали её в самом начале вместе с очередью). А при использовании методов без подчёркивания это должен сделать программист, так как эти методы должны выполняться с уже захваченной блокировкой. Подобный подход обеспечивает дополнительную гибкость при написании кода, но если механически заменить метод «с подчёркиванием» на «без подчёркивания» без

соответствующих изменений в коде, то это приведёт к мгновенному зависанию системы.

В этом и заключается классический подход. Мы разобрали его довольно поверхностно, так как подробные знания о том, как происходит обработка запросов с помощью очереди, не требуются для написания драйверов.

Запрос с обработкой вектора BIO

Следующий вариант (значение параметра mode равно 1) хотя и сложнее по смыслу, но проще с точки зрения реализации.

Каждая структура struct request (единичный блочный запрос в очереди) представляет собой один запрос ввода/вывода, хотя этот запрос может быть образован в результате слияния нескольких самостоятельных запросов, выполненного оптимизатором запросов, находящимся в ядре. Секторы, являющиеся результатом запроса, могут быть распределены по всей оперативной памяти получателя (т.е. представлять вектор ввода/вывода), но на блочном устройстве они всегда соответствуют набору последовательных секторов. Запрос (struct request) представлен в виде вектора сегментов, каждый из которых соответствует одному буферу в памяти (struct bio). Ядро может объединить несколько элементов вектора, связанных со смежными секторами на диске, в один элемент. Структура struct bio является низкоуровневым описанием части запроса блочного ввода/вывода (struct request), и этого уровня детализации вполне достаточно для создания драйверов.

В рамках этой стратегии функции драйвера должны будут выполнить последовательную обработку всех экземпляров struct bio, входящих в struct request, как показано в листинге 2.

Листинг 2. Обработка запросов с помощью вектора BIO

```
//функция для передачи одиночного BIO
static int xfer_bio( struct disk_dev *dev, struct bio *bio ) {
  int i, ret;
  struct bio_vec *bvec;
  sector_t sector = bio->bi_sector;
  DBG( "entering xfer_bio routine\n" );
  bio_for_each_segment( bvec, bio, i ) { //работаем с каждым сегментом
независимо.
  char *buffer = __bio_kmap_atomic( bio, i, KM_USER0 );
  sector_t nr_sectors = bio_sectors( bio );
  ret = transfer( dev, sector, nr_sectors, buffer, bio_data_dir( bio ) ==
WRITE );
  if( ret != 0 ) return ret;
  sector += nr_sectors;
    __bio_kunmap_atomic( bio, KM_USER0 );
  }
  return 0;
}
//функция для передачи всего запроса
static int xfer_request( struct disk_dev *dev, struct request *req ) {
  struct bio *bio;
  int nsect = 0;
  DBG( "entering xfer_request routine\n" );
  __rq_for_each_bio( bio, req ) {
    xfer_bio( dev, bio );
    nsect += bio->bi_size / KERNEL_SECTOR_SIZE;
  }
  return nsect;
}
//функция для обработки запроса с использованием вектора BIO
static void full_request( struct request_queue *q ) {
  struct request *req;
  int sectors_xferred;
  DBG( "entering full request routine\n" );
  req = blk_fetch_request( q );
  while( req ) {
    struct disk_dev *dev = req->rq_disk->private_data;
    if( !blk_fs_request( req )) {
      ERR( "skip non-fs request\n" );
      __blk_end_request_all( req, -EIO );
```

```
      req = blk_fetch_request( q );
      continue;
    }
    sectors_xferred = xfer_request( dev, req );
    if( !__blk_end_request_cur( req, 0 ) )
      req = blk_fetch_request( q );
  }
}
```

Код функции-обработчика full_request() похож на код функции simple_request() из предыдущего листинга, только вместо функции transfer() вызывается функция xfer_request(), последовательно извлекающая структуры struct bio из тела запроса и вызывающая функцию xfer_bio() для каждой такой структуры. Эта функция извлекает параметры и осуществляет обмен данными для единичного сегмента, используя функцию transfer() из листинга 1.

Прямое выполнение запроса без использования очереди

Очереди запросов в ядре реализуют интерфейс для подключения модулей, представляющих собой различные планировщики ввода/вывода. Задачей планировщика является упорядочение содержимого очереди так, чтобы предоставлять драйверу запросы ввода/вывода в последовательности , обеспечивающей максимальную производительность. Планировщик ввода/вывода также отвечает за объединение прилегающих запросов, так при получении нового запроса планировщик ищет в очереди запросы, относящиеся к прилегающим секторам, и если таковые нашлись, то они объединяются. При этом проверяются определённые условия, например, чтобы получившийся запрос не оказался слишком большим. В некоторых случаях, когда производительность не зависит от расположения секторов и

последовательности обращений (RAM-диски, флеш память, ...) можно отказаться от стандартного планировщика для очереди, заменив его на свою реализацию, которая будет немедленно получать запрос в момент его поступления.

Подобная стратегия, связанная с отказом от использования очереди, активируется, когда значение параметра mode равно 2. В этом случае создание и инициализация очереди в ядре для нашего устройства происходит совсем по другому алгоритму.

```
case RM_NOQUEUE:
  dev->queue = blk_alloc_queue( GFP_KERNEL );
  if( dev->queue == NULL ) goto out_vfree;
  blk_queue_make_request( dev->queue, make_request );
  break;
```

Для такой очереди необходимо установить свой обработчик для единичных сегментов (struct bio), требующих обслуживания (без оптимизации, слияний и т.п.):

```
static void make_request( struct request_queue *q, struct bio *bio ) {
  struct disk_dev *dev = q->queuedata;
  int status = xfer_bio( dev, bio );
  bio_endio( bio, status );
}
```

Функцию xfer_bio() можно найти в листинге 2. На этом мы завершим рассмотрение различных способов обработки запросов ввода-вывода в модуле блочного устройства.

Неприятным фактом является то, что API блочных устройств достаточно быстро меняется, и то, что было работоспособно в текущей версии ядра может не компилироваться в следующей.

Практическое тестирование

Мы закончим рассмотрение модулей блочных устройств

практическим тестированием созданных компонентов. Установим созданный модуль в систему:

```
# insmod block_mod_s.ko# ls -l /dev/x*
brw-rw---- 1 root disk 252,  0 нояб. 13 02:01 /dev/xda
...
brw-rw---- 1 root disk 252, 48 нояб. 13 02:01 /dev/xdd
# hdparm -g /dev/xdd
/dev/xdd:
 geometry      = 128/4/16, sectors = 8192, start = 16
```

Стандартным способом с помощью утилиты fdisk создадим некоторую структуру разделов:

```
# mkfs.vfat /dev/xdd
mkfs.vfat 3.0.12 (29 Oct 2011)
# fdisk -l /dev/xda
Диск /dev/xda: 4 МБ, 4194304 байт
4 heads, 16 sectors/track, 128 cylinders, всего 8192 секторов
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x2fb10b5b

Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/xda1            1        2000        1000   83  Linux
/dev/xda2         2001        5000        1500    5  Расширенный
/dev/xda3         5001        8191        1595+  83  Linux
/dev/xda5         2002        3500         749+  83  Linux
/dev/xda6         3502        5000         749+  83  Linux
# ls -l /dev/x*
brw-rw---- 1 root disk 252,  0 нояб. 13 02:11 /dev/xda
brw-rw---- 1 root disk 252,  1 нояб. 13 02:11 /dev/xda1
...
brw-rw---- 1 root disk 252,  6 нояб. 13 02:11 /dev/xda6
brw-rw---- 1 root disk 252, 16 нояб. 13 02:01 /dev/xdb
brw-rw---- 1 root disk 252, 32 нояб. 13 02:01 /dev/xdc
brw-rw---- 1 root disk 252, 48 нояб. 13 02:06 /dev/xdd
```

Как видно из начала вывода, диск /dev/xdd был отформатирован в формате FAT-32 без разметки на разделы. Теперь отформатируем один раздел на диске /dev/xda.

```
# mkfs.ext2 /dev/xda1
mke2fs 1.42.3 (14-May-2012)
...
# fsck /dev/xda1
fsck из util-linux 2.21.2
e2fsck 1.42.3 (14-May-2012)
/dev/xda1: clean, 11/128 files, 38/1000 blocks
```

```
...
# fsck /dev/xdd
fsck из util-linux 2.21.2
dosfsck 3.0.12, 29 Oct 2011, FAT32, LFN
/dev/xdd: 0 files, 0/2036 clusters
```

Мы можем смонтировать диски из числа отформатированных в заранее созданные каталоги:

```
# ls /mnt
dska  dskb  dskc  dskd  dske  efi  iso
# mount -tvfat /dev/xdd /mnt/dskd# mount -text2 /dev/xda1 /mnt/dska#
mount | grep /xd
/dev/xdd on /mnt/dskd type vfat
(rw,relatime,fmask=0022,dmask=0022,codepage=cp437,
iocharset=ascii,shortname=mixed,errors=remount-ro)
/dev/xda1 on /mnt/dska type ext2 (rw,relatime)
# df | grep /xd
/dev/xdd            4072          0    4072          0% /mnt/dskd
/dev/xda1            979         17     912          2% /mnt/dska
```

И проверить, как выполняется копирование файлов на созданных файловых системах.

```
# echo ++++++++++++++++++++ > /mnt/dska/f1# cp /mnt/dska/f1 /mnt/
dskd/f3# cat /mnt/dskd/f3
++++++++++++++++++++
# tree /mnt/dsk*
/mnt/dska
|-- f1
`-- lost+found
/mnt/dskd
`-- f3
2 directories, 3 files
```

А вот что показывает сравнительное тестирование скорости созданных дисковых устройств с помощью стандартной GNU-утилиты для разных выбранных стратегий обработки запросов ввода/вывода:

```
# insmod block_mod_s.ko mode=0# hdparm -t /dev/xda
/dev/xda:
 Timing buffered disk reads:   4 MB in  0.01 seconds = 318.32 MB/sec

# insmod block_mod_s.ko mode=1# hdparm -t /dev/xda
/dev/xda:
 Timing buffered disk reads:   4 MB in  0.03 seconds = 116.02 MB/sec
```

```
# insmod block_mod_s.ko mode=2# hdparm -t /dev/xda
/dev/xda:
 Timing buffered disk reads:   4 MB in  0.01 seconds = 371.92 MB/sec
```

Как видно, существуют весьма значительные различия в производительности разных подходов, но это предмет для совсем другого рассмотрения.