Модуль 4. Символьные устройства

Файлы символьных устройств

Цель этого занятия - написать драйвер символьного устройства. Мы будем разрабатывать драйвер символьного устройства, потому что этот класс драйверов предназначен для самых простых устройств. Символьные драйверы также легче понять, чем блочные или сетевые драйверы. Наша конечная цель заключается в написании примера модуля ядра - драйвера символьного устройства.

В качестве основы для привера воспользуемся фрагментами кода, извлечённого из реального драйвера устройства: scull (Simple Character Utility for Loading Localities, Простую Символьную Утилиту для Загрузки Местоположений, "череп", так же созвучно school, школа). scull является символьным драйвером, который оперирует с областью памяти, как будто это устройство. Далее будем в процессе проведения занятия будем использовать слово устройство наравне с "область памяти, используемая scull ", потому что это особенность scull.

Преимуществом scull является аппаратная независимость. scull просто работает с некоторой областью памяти, выделенной ядром. Любой пользователь может скомпилировать и запустить scull, и scull переносим на различные компьютерные архитектуры, на которых работает Linux. С другой стороны, устройство не делает ничего "полезного", кроме демонстрации интерфейса между ядром и символьными драйверами и возможности пользователю запускать некоторые тесты.

Первым шагом написания драйвера является определение возможностей (механизма), которые драйвер будет предлагать пользовательским программам. Так как наше "устройство" является частью памяти компьютера, мы свободны делать всё, что хотим. Это может быть устройство с последовательным или случайным доступом, одно устройство или много, и так далее.

Чтобы сделать scull полезным в качестве шаблона для написания настоящих драйверов для реальных устройств, в процессе занятий будет рассмотрено, как реализовать несколько абстракций устройств поверх памяти компьютера, каждая со своими особенностями.

Для начала реализуем поддержку драйвером файлов scull0 ... scull3. Четыре устройства, каждое содержит область памяти, которая одновременно и глобальная и стойкая. Глобальная означает, что если устройство открыто несколько раз, данные, содержащиеся в устройстве, является общими для всех файловых дескрипторов,

которые открыли его. Стойкое означает, что если устройство закрыть и вновь открыть, данные не потеряются. С этим устройством может быть интересно поработать, потому что оно может быть доступно и проверено с помощью обычных команд, таких как cp, cat и перенаправления ввода/вывода командной оболочки.

Символьные устройства доступны в файловой системе через имена. Эти имена называются специальными файлами или файлами устройств или просто узлами дерева файловой системы; они обычно находятся в каталоге /dev. Специальные файлы для символьных драйверов идентифицируются по “c” в первом столбце вывода по команде ls -l. Блочные устройства также представлены в /dev, но они идентифицируются по “b”. Сейчас мы будем говорить про символьные устройства, но большая часть следующей информации относится так же и к блочными устройствами.

Если вы введёте команду ls -l, то увидите два числа (разделённые запятой) в каждой записи файла устройства перед датой последней модификации файла, где обычно показывается длина. Эти цифры являются старшим и младшим номером устройства для каждого из них. Следующая распечатка показывает нескольких устройств, имеющихся в типичной системе. Их старшие номера: 1, 4, 7 и 10, а младшие: 1, 3, 5, 64, 65 и 129.

```
crw-rw-rw- 1 root root    1,  3 Apr 11 2002 null
crw------- 1 root root    10, 1 Apr 11 2002 psaux
crw------- 1 root root    4,  1 Oct 28 03:04 tty1
crw-rw-rw- 1 root tty     4, 64 Apr 11 2002 ttys0
crw-rw---- 1 root uucp    4, 65 Apr 11 2002 ttyS1
crw--w---- 1 vcsa tty     7,  1 Apr 11 2002 vcs1
crw--w---- 1 vcsa tty     7,129 Apr 11 2002 vcsa1
crw-rw-rw- 1 root root    1,  5 Apr 11 2002 zero
```

Традиционно, старший номер идентифицирует драйвер, ассоциированный с устройством. Например, и /dev/null и /dev/zero управляются драйвером 1, тогда как виртуальные консоли и последовательные терминалы управляются драйвером 4; аналогично, оба устройства vcs1 и vcsa1 управляются драйвером 7. Современные ядра Linux позволяют нескольким драйверам иметь одинаковые старшие номера, но большинство устройств, которые вы увидите, всё ещё организованы по принципу один-старший-один-драйвер.

Младший номер используется ядром, чтобы точно определить, о каком устройстве идёт речь. В зависимости от того, как написан драйвер (как мы увидим ниже), вы можете получить от ядра прямой указатель на своё устройство или вы можете использовать младший номер самостоятельно в качестве индекса в местном массиве устройств. В любом случае, само ядро почти ничего не знает о

младших номерах кроме того, что они относятся к устройствам, управляемым вашим драйвером.

Для хранения номеров устройств, обоих, старшего и младшего, в ядре используется тип dev_t (определённый в <linux/types.h>). Начиная с версии ядра 2.6.0, dev_t является 32-х разрядным, 12 бит отведены для старшего номера и 20 - для младшего. Ваш код, конечно, никогда не должен делать никаких предположений о внутренней организации номеров устройств; наоборот, он должен использовать набор макросов, находящихся в <linux/kdev_t.h> . Для получения старшей или младшей части dev_t используйте:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

Наоборот, если у вас есть старший и младший номера и необходимость превратить их в dev_t, используйте:

```
MKDEV(int major, int minor);
```

Заметим, что ядра версии 2.6 и старше могут вместить большое количество устройств, в то время как предыдущие версии ядра были ограничены 255-ю старшими и 255-ю младшими номерами. Предполагается, что такого широкого диапазона будет достаточно в течение довольно продолжительного времени, но компьютерная область достаточно усеяна ошибочными предположениями. Таким образом, вы должны ожидать, что формат dev_t может снова измениться в будущем; однако, если вы внимательно пишете свои драйверы, эти изменения не будут проблемой.

Одним из первых шагов, который необходимо сделать вашему драйверу при установке символьного устройства, является получение одного или нескольких номеров устройств для работы с ними. Необходимой функцией для выполнения этой задачи является register_chrdev_region, которая объявлена в <linux/fs.h>:

```
int register_chrdev_region(dev_t first, unsigned int
count, char *name);
```

Здесь first - это начало диапазона номеров устройств, который вы хотели бы выделить. Младшее число first часто 0, но не существует никаких требований на этот счёт. count - запрашиваемое общее число смежных номеров устройств. Заметим, что если число count большое, запрашиваемый диапазон может перекинуться на следующей старший номер, но всё будет работать правильно, если запрашиваемый диапазон чисел доступен. Наконец, name - это имя устройства, которое должно быть связано с этим диапазоном чисел; оно будет отображаться в /proc/devices и sysfs.

Как и в большинстве функций ядра, возвращаемое значение register_chrdev_region будет 0, если выделение была успешно выполнено. В случае ошибки будет возвращён отрицательный код

ошибки и вы не получите доступ к запрашиваемому региону. register_chrdev_region работает хорошо, если вы знаете заранее, какие именно номера устройств вы хотите. Однако, часто вы не будете знать, какие старшие номера устройств будут использоваться; есть постоянные усилия в рамках сообщества разработчиков ядра Linux перейти к использованию динамически выделяемых номеров устройств. Ядро будет счастливо выделить старший номер для вас "на лету", но вы должны запрашивать это распределение, используя другую функцию:

```
int alloc_chrdev_region(dev_t *dev, unsigned int
firstminor, unsigned int count, char *name);
```

В этой функции dev является только выходным значением, которое при успешном завершении содержит первый номер выделенного диапазона. firstminor должен иметь значение первого младшего номера для использования; как правило, 0. Параметры count и name аналогичны register_chrdev_region.

Независимо от того, как вы назначили номера устройств, вы должны освободить их, когда они больше не используются. Номера устройств освобождаются функцией:

```
void unregister_chrdev_region(dev_t first, unsigned int
count);
```

Обычное место для вызова unregister_chrdev_region будет в функции очистки вашего модуля.

Вышеприведённые функции выделяют номера устройств для использования вашим драйвером, но ничего не говорят ядру, что вы в действительности будете делать с этими номерами. Перед тем, как какая-либо программа из пространства пользователя сможет получить доступ к одному из этих номеров устройств, вашему драйверу необходимо подключить их к своим внутренним функциям, которые осуществляют операции устройства. Мы скоро будем описывать, как это осуществляется, но сначала рассмотрим динамическое выделение номеров устройств.

Некоторые старшие номера устройств для наиболее распространённых устройств выделены статически. Перечень этих устройств можно найти в Documentation/devices.txt в дереве исходных текстов ядра. Однако шансы, что статический номер уже был назначен перед использованием нового драйвера, малы и новые номера не назначаются (видимо, не будет назначен, если всё-таки совпадёт?). Так что, как у автора драйвера, у вас есть выбор: вы можете просто выбрать номер, который кажется неиспользованным, или вы можете определить старшие номера динамическим способом. Выбор номера может работать; пока вы единственный пользователь вашего драйвера; если ваш драйвер распространяется более широко,

случайно выбранный старший номер будет приводить к конфликтам и неприятностям.

Таким образом, для новых драйверов мы рекомендуем использовать динамическое выделение для получения старшего номера устройства, а не выбирать номер случайно из числа тех, которые в настоящее время свободны. Иными словами, ваш драйвер почти наверняка должен использовать alloc_chrdev_region вместо register_chrdev_region.

Недостатком динамического назначения является то, что вы не сможете создать узлы устройств заранее, так как старший номер, выделяемый для вашего модуля, будет меняться. Вряд ли это проблема для нормального использования драйвера, потому что после того, как номер был назначен, вы можете прочитать его из /proc/devices.

Следовательно, чтобы загрузить драйвер, использующий динамический старший номер, вызов insmod может быть заменён простым скриптом, который после вызова insmod читает /proc/devices в целью создания специального файла (ов).

Типичный файл /proc/devices выглядит следующим образом:

Символьные устройства:
1 mem
2 pty
3 ttyp
4 ttyS
6 lp
7 vcs
10 misc
13 input
14 sound
21sg
180 usb

Блочные устройства:
2 fd
8 sd
11sr
65 sd
66 sd

Следовательно, чтобы извлечь информацию из /proc/devices для создания файлов в каталоге /dev, скрипт загрузки модуля, которому был присвоен динамический номер, может быть написан с использованием такого инструмента, как awk.

Следующий скрипт, scull_load, является частью дистрибутива scull. Пользователь драйвера, который распространяется в виде модуля, может вызывать такой сценарий из системного файла rc.local или запускать его вручную каждый раз, когда модуль становится необходим.

```
#!/bin/sh module="scull" device="scull" mode="664"
# вызвать insmod со всеми полученными параметрами
# и использовать имя пути, так как новые modutils не
просматривают . по умолчанию
/sbin/insmod ./$module.ko $* || exit 1

# удалить давно ненужные узлы rm -f /dev/${device}[0-3]

major=$(awk "\$2==\"$module\" {print \$1}" /proc/
devices)

mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3

# назначьте соответствующую группу/разрешения, и
измените группу.
# не все дистрибутивы имеют "staff", некоторые вместо
этого используют "wheel".

group="staff"
grep -q '^staff:' /etc/group || group="wheel"
chgrp $group /dev/${device}[0-3]
chmod $mode /dev/${device}[0-3]
```

Скрипт может быть адаптирован для других драйверов путём переопределения переменных и корректировке строчек с mknod. Скрипт просто показывает создание четырёх устройств, потому что в исходниках scull по умолчанию используется число четыре.

Последние несколько строчек скрипта могут показаться неясными: зачем менять группы и режим работы устройства? Причина в том, что скрипт должен запускаться суперпользователем (superuser), так что вновь созданные специальные файлы принадлежат root-у. По умолчанию биты разрешения установлены так, что только root имеет доступ на запись, остальные могут получать доступ на чтение. Как правило, узлы устройств требуют иной политики доступа, так что тем или иным путём права доступа должны быть изменены.

Для очистки каталога /dev и удаления модуля так же доступен скрипт scull_unload.

Если неоднократное создание и уничтожение узлов /dev выглядит как излишество, есть полезный обходной путь. Если вы загружаете и выгружаете только один драйвер, после первого создания специальных файлов вашим скриптом вы можете просто использовать rmmod и insmod: динамические номера не случайны (не рандомизированы) и вы можете рассчитывать, что такие же номера выбираются каждый раз, если вы не загружаете какие-то другие (динамические) модули. Избегание больших скриптов полезно в процессе разработки. Но очевидно, что этот трюк не масштабируется более чем на один драйвер за раз.

Лучшим способом присвоения старших номеров, на наш взгляд, является использование по умолчанию динамического присвоения, оставив себя возможность указать старший номер во время загрузки или даже во время компиляции. scull выполняет работу таким образом; он использует глобальную переменную scull_major, чтобы сохранить выбранный номер (есть также scull_minor для младшего номера). Переменная инициализируется SCULL_MAJOR, определённым в scull.h. Значение по умолчанию SCULL_MAJOR в распространяемых исходниках равно 0, что означает "использовать динамическое определение". Пользователь может принять значение по умолчанию или выбрать специфичный старший номер либо изменив макрос перед компиляцией, либо указав значение для scull_major в командной строки insmod. Наконец, с помощью скрипта scull_load пользователь может передать аргументы insmod в командной строке scull_load.

Для получения старшего номера в исходниках scull используется такой код:

```
if (scull_major) {
dev = MKDEV(scull_major, scull_minor);
result = register_chrdev_region(dev, scull_nr_devs,
«scull");
} else {
result = alloc_chrdev_region(&dev, scull_minor,
scull_nr_devs, "scull"); scull_major = MAJOR(dev);
}
if (result < 0) {
printk(KERN_WARNING "scull: can't get major %d\n",
scull_major);
return result; }
```

Структура file_operations

Регистрация номера устройства является лишь первой из многих задач, которые должен выполнять код драйвера. Мы рассмотрим другие важные компоненты драйвера чуть позже, но одно отступление необходимо в первую очередь. Большинство основных операций драйвера включает три важных структуры данных ядра, называемые file_operations, file и inode. Требуется базовое знакомство с этими структурами, чтобы быть способным делать много всего интересного, так что сейчас мы бросим быстрый взгляд на каждую из них, прежде чем углубляться в подробности того, как осуществляются основные операции драйвера.

Ранее мы рассмотрели резервирование номеров устройств для нашего использования, но мы ещё не подключили никаких своих драйверных операций на эти номера. Структура file_operations определяет, как символьный драйвер устанавливает эту связь. Структура определена в <linux/fs.h>, это коллекция указателей на функции. Каждое открытие файла (представленное внутри структуры file, которую мы рассмотрим далее) связано с собственным набором функций (включая поле, названное f_op, которое указывает на структуру file_operations). Операции в основном отвечают за осуществление системных вызовов и, таким образом, названы open (открыть), read (читать) и так далее. Мы можем рассматривать файл как "объект" и функции, работающие с ним, будут его "методами", используя терминологию объектно-ориентированного программирования для обозначения действий, декларируемых исполняющим их объектом. Это первый признак объектно-ориентированного программирования, видимого нами в ядре Linux, и мы увидим подобные примеры и позже.

Обычно, структура file_operations или указатель на неё называется fops (или как-то похоже). Каждое поле структуры должно указывать на функцию в драйвере, которая реализует соответствующие операции, или оставить NULL для неподдерживаемых операций. Точное поведение ядра, когда задан указатель NULL, отличается для каждой функции, список показан позже в этом разделе.

В приведенном ниже списке приводятся все операции, которые приложение может вызывать на устройстве. Мы постарались сохранить список кратким, поэтому он может быть использован в качестве справки, только кратко описывающим каждую операцию и поведение ядра по умолчанию, когда используется указатель NULL.

Прочитав список методов file_operations, вы заметите, что некоторое число параметров включает строку __user. Эта аннотация является

одной из форм документации, отмечая, что указатель является адресом пространства пользователя, который не может быть разыменован непосредственно. Для нормальной компиляции __user не имеет никакого эффекта, но может быть использована внешним проверяющим программным обеспечением, чтобы найти злоупотребление адресами пользовательского пространства.

`struct module *owner`

Первое поле file_operations вовсе не операция, это указатель на модуль, который "владеет" структурой. Это поле используется для предотвращения выгрузки модуля во время использования его операций. Почти всегда оно просто инициализируется THIS_MODULE, макрос определён в <linux/module.h>.

```
loff_t (*llseek) (struct file *, loff_t, int);
```

Метод llseek используется для изменения текущей позиции чтения/записи в файле, а новая позиция передаётся как (положительное) возвращаемое значение. Параметром loff_t является "длинное смещение" и он, по крайней мере, 64 бита даже на 32-х разрядных платформах. Об ошибках сигнализируют отрицательные возвращаемые значения. Если указатель на эту функцию NULL, вызовы поиска позиции будут изменять счётчик позиции в структуре file (описанной в разделе "Структура file" 50) потенциально непредсказуемым образом.

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Используется для получения данных от устройства. Указатель NULL в этой позиции заставляет системный вызов read вернуться с ошибкой -EINVAL (“Invalid argument”, "Недопустимый аргумент"). Неотрицательное возвращаемое значение представляет собой число успешно считанных байт (возвращаемое значение является типом "размер со знаком", обычно родным (нативным) целочисленным типом для целевой платформы).

```
ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);
```

Начинает асинхронное чтение - операцию чтения, которая может быть не завершена перед возвратом из функции. Если этот метод NULL, все операции будут обрабатываться (синхронно) вместо неё функцией read.

```
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

Отправляет данные на устройство. Если NULL, -EINVAL возвращается в программу, сделавшую системный вызов write. Возвращаемое значение, если оно неотрицательное, представляет собой число успешно записанных байт.

```
ssize_t (*aio_write)(struct kiocb *, const char __user *,
size_t, loff_t *);
```

Начинает асинхронную операции записи на устройство.

```
int (*readdir) (struct file *, void *, filldir_t);
```

Это поле должно быть NULL для файлов устройства; оно используется для чтения каталогов и используется только для файловых систем.

```
unsigned int (*poll) (struct file *, struct
poll_table_struct *);
```

Метод poll (опрос) является внутренним для трёх системных вызовов: poll, epoll и select, все они используются для запросов чтения или записи, чтобы заблокировать один или несколько файловых дескрипторов. Метод poll должен вернуть битовую маску, показывающую, возможны ли неблокирующие чтение или запись, и, возможно, предоставить ядру информацию, которая может быть использована вызывающим процессом для ожидания, когда ввод/вывод станет возможным. Если драйвер оставляет метод poll NULL, предполагается, что устройство читаемо и записываемо без блокировки.

```
int (*ioctl) (struct inode *, struct file *, unsigned
int, unsigned long);
```

Системный вызов ioctl (управление вводом-выводом) открывает путь к выполнению зависящих от устройства команд (например, форматирование дорожки гибкого диска, которая и не чтение, и не запись). Кроме того, несколько команд ioctl распознаются ядром без ссылки на таблицу fops. Если устройство не обеспечивает метод ioctl, системный вызов возвращает ошибку на любой запрос, который не является предопределенным (-ENOTTY, “No such ioctl for device”, "Нет такого управления вводом- выводом для устройства").

```
int (*mmap) (struct file *, struct vm_area_struct *);
```

mmap (отобразить в память) используется, чтобы запросить отображение памяти устройства на адресное пространство процесса. Если этот метод NULL, системный вызов mmap возвращает -ENODEV.

```
int (*open) (struct inode *, struct file *);
```

Хотя это всегда первая операция, выполняющаяся с файлом устройства, драйверу не требуется декларировать соответствующий метод. Если этот параметр NULL, открытие устройства всегда успешно, но ваш драйвер не уведомляется.

```
int (*flush) (struct file *);
```

Операция flush (сбросить на диск) вызывается, когда процесс закрывает свою копию файла дескриптора устройства; она должна выполнить (и ожидает это) любую ожидающую выполнения

операцию на устройстве. Не следует путать это с операцией fsync, запрашиваемой программами пользователя. В настоящее время flush используется в очень редких драйверах; например, драйвер ленточного накопителя SCSI использует её, чтобы гарантировать, что все данные записаны на ленту, прежде чем устройство закроется. Если flush NULL, ядро просто игнорирует запрос пользовательского приложения.

```
int (*release) (struct inode *, struct file *);
```

Эта операция (отключение) вызывается, когда файловая структура освобождается. Как и open, release может быть NULL. (* Обратите внимание, что release не вызывается каждый раз, когда процесс вызывает close. Когда файловая структура используется несколькими процессами (например, после fork или dup), release не будет вызван, пока не будут закрыты все копии. Если вам необходимо при завершении копирования сбросить на диск ожидающие данные, вы должны реализовать метод flush.)

```
int (*fsync) (struct file *, struct dentry *, int);
```

Этот метод является внутренним системным вызовом fsync, который пользователь вызывает для сброса на диск любых ожидающих данных. Если этот указатель NULL, системный вызов возвращает -EINVAL.

```
int (*aio_fsync)(struct kiocb *, int);
```

Это асинхронная версия метода fsync.

```
int (*fasync) (int, struct file *, int);
```

Эта операция используется, чтобы уведомить устройство, изменив его флаг FASYNC. Асинхронные уведомления - сложная тема. Поле может быть NULL, если драйвер не поддерживает асинхронные уведомления.

```
int (*lock) (struct file *, int, struct file_lock *);
```

Метод lock (блокировка) используется для реализации блокировки файла; блокировка является неотъемлемой функцией для обычных файлов, но почти никогда не реализуется драйверами устройств.

```
ssize_t (*readv) (struct file *, const struct iovec *,
unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *,
unsigned long, loff_t *);
```

Эти методы осуществляют разбросанные (scatter/gather, разборка/сборка) операции чтения и записи. Приложениям иногда необходимо сделать одну операцию чтения или записи с участием нескольких областей памяти; эти системные вызовы позволяют им сделать это без необходимости дополнительных операций копирования данных. Если эти указатели на функции остаются

NULL, вместо них вызываются методы read и write (возможно, несколько раз).

```
ssize_t (*sendfile)(struct file *, loff_t *, size_t,
read_actor_t, void *);
```

Этот метод реализует читающую сторону системного вызова sendfile (послать файл), который перемещает данные из одного файлового дескриптора на другой с минимумом копирования. Используется, например, веб-сервером, которому необходимо отправить содержимое файла из сети наружу. Драйверы устройств обычно оставляют sendfile NULL.

```
ssize_t (*sendpage) (struct file *, struct page *, int,
size_t, loff_t *, int);
```

sendpage (послать страницу) - это другая половина sendfile, она вызывается ядром для передачи данных, одну страницу за один раз, в соответствующий файл. Драйверы устройств, как правило, не реализуют sendpage.

```
unsigned long (*get_unmapped_area)(struct file *,
unsigned long, unsigned long, unsigned long, unsigned
long);
```

Цель этого метода - найти подходящее место в адресном пространстве процесса, чтобы отобразить в сегменте памяти нижележащее устройство. Эта задача обычно выполняется кодом управления памятью; этот метод существует, чтобы разрешить драйверу реализовать любое согласование требований, которое может иметь специфичное устройство. Большинство драйверов может оставить этот метод NULL.

```
int (*check_flags)(int)
```

Этот метод позволяет модулю проверить флаги, передаваемые вызову fcntl(F_SETFL...).

```
int (*dir_notify)(struct file *, unsigned long);
```

Этот метод вызывается, когда приложение использует fcntl для запроса уведомлений об изменении директории. Это полезно только для файловых систем; драйверам нет необходимости реализовывать dir_notify.

Драйвер устройства scull реализует только самые важные методы устройства. Его структура file_operations инициализируется следующим образом:

```
struct file_operations scull_fops = {
    .owner = THIS_MODULE,
    .llseek = scull_llseek,
    .read = scull_read,
    .write = scull_write,
    .ioctl = scull_ioctl,
```

```
    .open = scull_open,
    .release = scull_release,
};
```

Эта декларация использует синтаксис инициализации стандартной маркированной структуры языка Си. Этот синтаксис является предпочтительным, поскольку делает драйвера более переносимыми через изменения в определениях структуры и, возможно, делает код более компактным и читабельным.

Маркированная инициализация разрешает переназначение членов структуры; в некоторых случаях были реализованы существенные улучшения производительности путём размещения указателей к наиболее часто используемым членам в одной строке кэша оборудования.

Структура file

Структура file, определённая в <linux/fs.h>, является второй наиболее важной структурой данных, используемой драйверами устройств. Обратите внимание, что file не имеет ничего общего с указателями FILE программ пространства пользователя. FILE определён в библиотеке Си и никогда не появляется в коде ядра. Структура file, с другой стороны, это структура ядра, которая никогда не появляется в пользовательских программах.

Структура file представляет открытый файл. (Это не специфично для драйверов устройств; каждый открытый файл в системе имеет ассоциированную структуру file в пространстве ядра.) Она создаётся ядром при открытии и передаётся в любую функцию, которая работает с файлом, до последнего закрытия. После закрытия всех экземпляров файла ядро освобождает структуру данных.

В исходных текстах ядра указатель на структуру file обычно назван или file или filp ("указатель на файл"). Мы будем использовать название указателя filp для предотвращения путаницы с самой структурой. Таким образом, file относится к структуре, а filp - указатель на структуру.

Здесь показаны наиболее важные поля структуры file. Как и в предыдущем разделе, список может быть пропущен при первом чтении. Тем не менее, когда мы столкнёмся с некоторым реальным кодом Си, мы обсудим эти поля более подробно.

```
mode_t f_mode;
```

Определяет режим файла как или читаемый или записываемый (или и то и другое) с помощью битов FMODE_READ и FMODE_WRITE. Вы можете захотеть проверить это поле для разрешения чтения/записи в своей функции open или ioctl, но вам не нужно проверять разрешения для read и write, потому что ядро проверяет это перед вызовом вашего метода. Попытка чтения или записи, если файл не был открыт для этого типа доступа, отклоняется, так что драйвер даже не узнаёт об этом.

```
loff_t f_pos;
```

Текущая позиция чтения или записи. loff_t является 64-х разрядным значением на всех платформах (long long в терминологии gcc). Драйвер может читать это значение, чтобы узнать текущую позицию в файле, но обычно не должен изменять его; read и write должны обновить позицию с помощью указателя, который они получают в качестве последнего аргумента, вместо воздействия на filp->f_pos напрямую. Единственным исключением из этого правила является метод llseek, целью которого является изменение позиции файла.

```
unsigned int f_flags;
```

Существуют флаги файлов, такие как O_RDONLY, O_NONBLOCK и O_SYNC. Драйвер должен проверить флаг O_NONBLOCK, чтобы увидеть, была ли запрошена неблокирующая операция; другие флаги используются редко. В частности, разрешение на чтение/запись должно быть проверено с помощью f_mode, а не f_flags. Все флаги определены в заголовке <linux/fcntl.h>.

`struct file_operations *f_op;`

Операции, связанные с файлом. Ядро присваивает указатель как часть своей реализации open, а затем считывает его, когда необходимо выполнить любые операции. Значение в filp->f_op никогда не сохраняется ядром для дальнейшего использования; это означает, что вы можете изменять файловые операции, связанные с вашим файлом, и новые методы будут действовать после вашего возвращения к вызвавшей программе. Например, код для open, связанный со старшим номером 1 (/dev/null, /dev/zero, и так далее), заменяет операции в filp->f_op в зависимости от открытого младшего номера. Эта практика позволяет реализовать несколько поведений под тем же основным номером без дополнительных накладных расходов при каждом системном вызове. Возможность заменить файловые операции является в ядре эквивалентом "переопределения метода" в объектно-ориентированном программировании.

`void *private_data;`

Системный вызов open устанавливает этот указатель в NULL для драйвера перед вызовом метода open. Вы можете сделать своё собственное использование поля или игнорировать его; вы можете использовать поле как указатель на выделенные данные, но тогда вы должны помнить об освобождении памяти в методе release до уничтожения структуры file ядром. private_data является полезным ресурсом для сохранения информации о состоянии через системные вызовы и используется в большинстве наших примеров модулей.

`struct dentry *f_dentry;`

Структура элемента каталога (directory entry, dentry), связанного с файлом. Авторам драйверов устройств обычно не требуется заботиться о структуре dentry, помимо доступа к структуре inode через filp->f_dentry->d_inode.

Реальная структура имеет несколько больше полей, но они не являются полезными для драйверов устройств. Мы можем смело игнорировать эти поля, поскольку драйверы никогда не создают структуры файлов; они только обращаются к структурам, созданным другими.

Регистрация устройства

Структура **inode** (индексный дескриптор) используется ядром внутренне для представления файлов. Поэтому она отличается от файловой структуры, которая представляет открытый файловый дескриптор. Может быть большое число файловых структур, представляющих собой множество открытых дескрипторов одного файла, но все они указывают на единственную структуру inode.

Структура inode содержит большое количество информации о файле. По общему правилу только два поля этой структуры представляют интерес для написания кода драйвера:

```
dev_t i_rdev;
```

Это поле содержит фактический номер устройства для индексных дескрипторов, которые представляют файлы устройств.

```
struct cdev *i_cdev;
```

Структура cdev является внутренней структурой ядра, которая представляет символьные устройства; это поле содержит указатель на ту структуру, где inode ссылается на файл символьного устройства.

Тип i_rdev изменился в течение серии разработки ядра версии 2.5, поломав множество драйверов. В качестве средства поощрения более переносимого программирования разработчики ядра добавили два макроса, которые могут быть использованы для получения старшего и младшего номера inode:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

В интересах не быть пойманными следующими изменениями, вместо манипулирования i_rdev напрямую должны быть использованы эти макросы.

Как уже упоминалось, ядро использует структуры типа cdev для внутреннего представления символьных устройств. Перед тем, как ядро вызовет операции устройства, вы должны выделить и зарегистрировать одну или больше этих структур. Существует старый механизм, который избегает применения структур cdev которые мы обсудим в разделе "Старый способ". Однако, новый код должен использовать новую технику. Чтобы сделать это, ваш код должен подключить <linux/cdev.h>, где определены структура и связанные с ней вспомогательные функции. Есть два способа создания и инициализации каждой из этих структур. Если вы хотите получить автономную структуру cdev во время выполнения, вы можете сделать это таким кодом:

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

Однако, скорее всего, вы захотите вставлять свои собственные устройство-зависимые структуры cdev; это то, что делает scull. В этом случае вы должны проинициализировать те структуры, что уже созданы с помощью:

```
void cdev_init(struct cdev *cdev, struct
file_operations *fops);
```

В любом случае есть ещё одно поле структуры cdev, которое необходимо проинициализировать. Так же как в структуре file_operations, структура cdev имеет поле owner (владелец), которое должно быть установлено в THIS_MODULE.

Последний шаг после создания структуры cdev - сказать об этом ядру вызовом:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned
int count);
```

Здесь, dev является структурой cdev, num - первый номер устройства, на который реагирует данное устройство, а count - количество номеров устройств, которые должны быть связаны с устройством. Часто count единица, но бывают ситуации, когда имеет смысл иметь более одного номера устройства, соответствующего определённому устройству. Рассмотрим, например, драйвер ленточного SCSI накопителя, который позволяет пользовательскому пространству выбирать режимы работы (например, плотность) путём назначения нескольких младших номеров для каждого физического устройства.

При использовании cdev_add необходимо иметь в виду несколько важных вещей. Первым является то, что этот вызов может потерпеть неудачу. Если он вернул код ошибки, ваше устройство не было добавлено в систему. Однако, это почти всегда удаётся, что вызывает другой момент: как только cdev_add возвращается, устройство становится "живым" и его операции могут быть вызваны ядром. Вы не должны вызывать cdev_add, пока драйвер не готов полностью проводить операции на устройстве.

Чтобы удалить символьное устройство из системы, вызовите:

```
void cdev_del(struct cdev *dev);
```

Очевидно, что вы не должны обращаться к структуре cdev после передачи её в cdev_del.

Внутри scull представляет каждое устройство структурой типа scull_dev. Эта структура определена как:

```
struct scull_dev {
/* Указатель, установленный на первый квант */
  struct scull_qset *data;
/* размер текущего кванта */
```

```
    int quantum;
  /* размер текущего массива */
    int qset;
  /* количество данных, хранимых здесь */
    unsigned long size;
  /* используется sculluid и scullpriv */
    unsigned int access_key;
  /* семафор взаимного исключения */
    struct semaphore sem;
  /* структура символьного устройства */
    struct cdev cdev;
  };
```

Мы обсудим различные поля этой структуры, когда подойдём к ним, но сейчас мы обратим внимание на cdev, структуру типа cdev, которая является интерфейсами нашего устройства к ядру. Эта структура должна быть проинициализирована и добавлена в систему, как описано выше; код scull, который решает эту задачу:

```
static void scull_setup_cdev(struct scull_dev *dev, int
index)
{
    int err, devno = MKDEV(scull_major, scull_minor +
index);
    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1);
/* Терпите неудачу изящно, если это необходимо */
    if (err)
        printk(KERN_NOTICE "Error %d adding scull%d",
err, index);
}
```

Поскольку структуры cdev внедрены в структуру scull_dev, чтобы выполнить инициализацию этой структуры должна быть вызвана cdev_init .

Старый способ

Если вы пороетесь во многих драйверных кодах в ядре, то сможете заметить, что довольно много символьных драйверов не используют интерфейс cdev, который мы только что описали. То, что вы видите, является старым кодом. Так как этот код работает, это обновление может не произойти в течение длительного времени. Для полноты картины мы описываем старый интерфейс регистрации

символьных устройств, но новый код не должен его использовать; этот механизм, вероятно, уйдёт в будущем ядре.

Классический способ зарегистрировать символьный драйвер устройства:

```
int register_chrdev(unsigned int major, const char
*name, struct file_operations *fops);
```

Здесь major является запрашиваемым старшим номером, name - это имя драйвера (оно появляется в /proc/devices) и fops является структурой по умолчанию file_operations. Вызов register_chrdev регистрирует младшие номера 0 - 255 для данного старшего и устанавливает для каждого структуру по умолчанию cdev. Драйверы, использующие этот интерфейс, должны быть готовы для обработки вызовов open по всем 256 младшим номерам (независимо от того, соответствуют ли они реальным устройствам или нет) и они не могут использовать старший или младший номера больше 255.

Чтобы сделать любые инициализации при подготовке к поздним операциям для драйвера, предусмотрен метод open. В большинстве драйверов open должен выполнять следующие задачи:

- Проверку зависимых от устройства ошибок (таких, как "устройство не готово" или аналогичных проблем с оборудованием);
- Проинициализировать устройство, если оно открывается в первый раз;
- Обновить в случае необходимости указатель f_op;
- Создать и заполнить любые структуры данных для размещения в filp->private_data;

Первым делом, однако, обычно определяется, какое устройство открывается в настоящий момент. Вспомните прототип метода open:

```
int (*open) (struct inode *, struct file *);
```

Аргумент inode имеет информацию, которая нам необходима, в форме его поля i_cdev, в котором содержится структура cdev, которую мы создали раньше. Единственной проблемой является то, что обычно мы не хотим саму структуру cdev, мы хотим структуру scull_dev, которая содержит эту структуру cdev. Язык Си позволяет программистам использовать всевозможные трюки, чтобы выполнить такое преобразование, однако, программирование таких трюков чревато ошибками и приводит к коду, который труден для чтения и понимания другими. К счастью, в данном случае программисты ядра сделали сложную работу за нас, в виде макроса container_of, определённого в <linux/kernel.h>:

```
container_of(pointer, container_type, container_field);
```

Этот макрос получает указатель на поле под названием container_field, находящееся внутри структуры типа container_type, и возвращает указатель на эту структуру. В scull_open этот макрос используется, чтобы найти соответствующую структуру устройства:

```
struct scull_dev *dev; /* информация об устройстве */
dev = container_of(inode->i_cdev, struct scull_dev,
cdev);
filp->private_data = dev; /* для других методов */
```

Как только он нашел структуру scull_dev, для облегчения доступа в будущем scull сохраняет указатель на неё в поле private_data структуры file.

Другим способом идентификации открываемого устройства является поиск младшего номера, сохранённого в структуре inode. Если вы регистрируете устройство с register_chrdev, вы должны использовать эту технику. Обязательно используйте iminor, чтобы получить младший номер из структуры inode, а также убедиться, что он соответствует устройству, которое ваш драйвер действительно готов обслужить.

(Немного упрощённый) код для scull_open:

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* информация об устройстве */
    dev = container_of(inode->i_cdev, struct scull_dev,
cdev);
    filp->private_data = dev; /* для других методов */
/* теперь установим в 0 длину устройства, если открытие
было только для записи */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY)
    {
        scull_trim(dev); /* игнорирование ошибок */
    }
    return 0; /* успешно */
}
```

Код выглядит очень небольшим, поскольку не делает какого-либо обращения к определённому устройству, когда вызывается open. В этом нет необходимости, потому что устройство scull по дизайну является глобальным и стойким. В частности, нет никаких действий, таких как "инициализация устройств при первом открытии", поэтому мы не хранимсчётчик открытий для устройств scull.

Единственная реальная операция, выполняемая на устройстве, это усечение его длины до 0, когда устройство открывается для записи. Это выполняется, так как по дизайну перезапись устройства scull более коротким файлом образует более короткую область данных

устройства. Это подобно тому, как открытие обычного файла для записи обрезает его до нулевой длины. Операция ничего не делает, если устройство открыто для чтения.

Отключение устройства

Роль метода release обратна open. Иногда вы обнаружите, что реализация метода называется device_close вместо device_release. В любом случае, метод устройства должен выполнять следующие задачи:

- Освободить всё, что open разместил в filp->private_data;
- Выключить устройство при последнем закрытии;

Базовая форма scull не работает с аппаратурой, которую надо выключать, так что код необходим минимальный:

```
int scull_release(struct inode *inode, struct file *filp)
{
    return 0;
}
```

Вы можете быть удивлены, что происходит, когда файл устройства закрывается большее число раз, чем открывается. В конце концов, системные вызовы dup и fork создают копии открытых файлов без вызова open; каждая из этих копий затем закрывается при завершении программы. Например, большинство программ не открывают свой файл stdin (или устройство), но все они в конечном итоге его закрывают. Как драйвер узнает, когда открытый файл устройства действительно был закрыт?

Ответ прост: не каждый системный вызов close вызывает метод release. Только вызовы, которые действительно освобождают структуру данных устройства, запускают этот метод, отсюда его название. Ядро хранит счётчик, сколько раз используется структура file. Ни fork, ни dup не создают новую структуру file (только open делает это); они просто увеличивают счётчик в существующей структуре. Системный вызов call запускает метод release только тогда, когда счётчик для структуры file падает до 0, что происходит, когда структура уничтожена. Эта взаимосвязь между методом release и системным вызовом close гарантирует, что ваш драйвер видит только один вызов release для каждого open.

Обратите внимание, что метод flush вызывается каждый раз, когда приложение вызывает close. Однако, очень немногие драйверы реализуют flush, потому что обычно нечего делать во время закрытия, пока не вызван release.

Как вы можете себе представить, предыдущее обсуждение применяется даже тогда, когда приложение завершается без явного закрытия открытых файлов: ядро автоматически закрывает все файлы во время завершения процесса внутренне с помощью системного вызова close.

Пример драйвера символьного устройства с доступом только на чтение
Специфика поддержки записи для символьных устройств
Пример драйвера символьного устройства с доступом на чтение и запись

Перед знакомством с операциями read и write мы рассмотрим получше, как и почему scull выполняет выделение памяти. "Как" необходимо, чтобы глубоко понимать код, а "почему" демонстрирует варианты выбора, который должен делать автор драйвера, хотя scull, безусловно, не типичен, как устройство.

Этот раздел имеет дело только с политикой распределения памяти в scull и не показывает навыки аппаратного управления, необходимые для написания реальных драйверов.

Область памяти, используемая scull, также называемая устройством, имеет переменную длину. Чем больше вы пишете, тем больше она растёт; укорачивание производится перезаписью устройства более коротким файлом.

Драйвер scull знакомит с двумя основными функциями, используемыми для управления памятью в ядре Linux. Вот эти функции, определённые в <linux/slab.h>:

```
void *kmalloc(size_t size, int flags);
void kfree(void *ptr);
```

Вызов kmalloc пытается выделить size байт памяти; возвращаемая величина - указатель на эту память или NULL, если выделение не удаётся. Аргумент flags используется, чтобы описать, как должна быть выделена память; мы изучим эти флаги позже. Сейчас мы всегда используем GFP_KERNEL. Выделенная память должна быть освобождена kfree. Вы никогда не должны передавать kfree что-то, что не было получено от kmalloc. Однако, правомерно передать kfree указатель NULL.

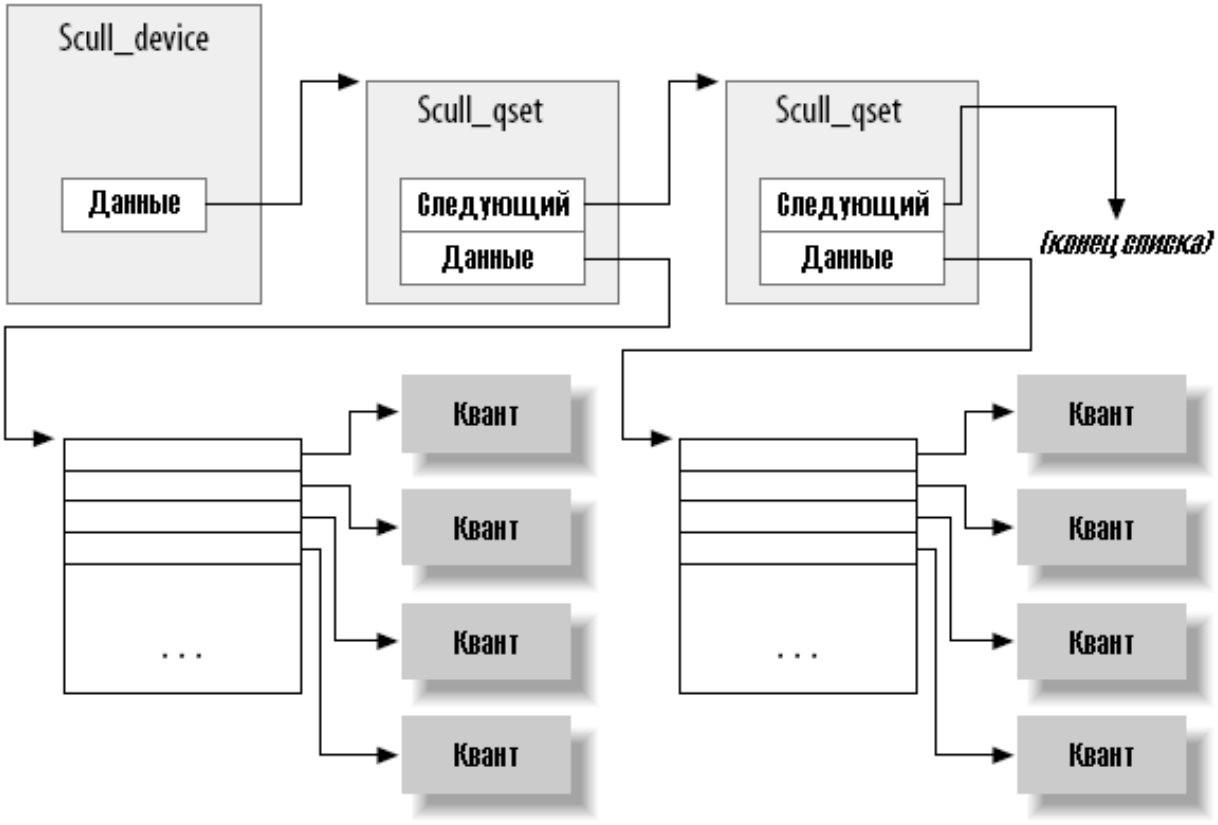
kmalloc это не самый эффективный способ распределения больших областей памяти, поэтому сделанный для scull выбор не особенно умный. Исходный код для изящной реализации будет более трудным для восприятия, а целью этого раздела является показать read и write, а не управление памятью. Вот почему код просто использует kmalloc и kfree, без пересортировки выделенных целых страниц, хотя такой подход был бы более эффективным.

С другой стороны, мы не хотим ограничивать размер области "устройства" по философским и практическим соображениям.

Философски, это всегда плохая идея - поставить произвольные ограничения на управляемые объекты данных. Практически, scull может быть использован для временного поедания всей памяти вашей системы в целях выполнения тестов в условиях малого количества памяти. Выполнение таких тестов могло бы помочь вам понять внутренности системы. Вы можете использовать команду cp /dev/zero / dev/scull0, чтобы съесть всю реальную оперативную память с помощью scull, и вы можете использовать утилиту dd, чтобы выбрать, какой объём данных скопируется на устройство scull.

В scull каждое устройство представляет собой связный список указателей, каждый из которых указывает на структуру scull_qset. По умолчанию каждая такая структура может ссылаться на более чем четыре миллиона байт через массив промежуточных указателей.

Реализованный исходник использует массив из 1000 указателей на области по 4000 байта. Мы называем каждую область памяти квантом (quantum), а массив (или его длину) - набором квантов (quantum set). Устройство scull и его области памяти показаны на рисунке:



Выбранные числа таковы, что запись одного байта в scull потребляет 8000 или 12000 байт памяти: 4000 для кванта и 4000 или 8000 для набора квантов (в зависимости от того, является ли на целевой платформе указатель 32-х разрядным или 64-х разрядным). Если, наоборот, вы пишете большое количество данных, накладные

расходы на связный список не такие большие. Существует только один список элементов для каждых четырёх мегабайт данных и максимальный размер устройства ограничен объёмом памяти компьютера.

Выбор соответствующего значения для кванта и квантового набора - это вопрос политики, а не механизма, и их оптимальные размеры зависят от того, как используется устройство. Таким образом, драйвер scull не должен заставлять использовать какие-то определённые значения для размеров кванта и набора квантов. В scull пользователь может изменять эти значения несколькими способами: путём изменения макросов SCULL_QUANTUM и SCULL_QSET в scull.h во время компиляции, устанавливая целочисленные значения scull_quantum и scull_qset, во время загрузки модуля, или изменяя текущее значение и значение по умолчанию с помощью ioctl во время выполнения. Использование макроса и целой величины позволяют выполнять конфигурацию и во время компиляции и во время загрузки и напоминают выбор старшего номера. Мы используем эту технику для любого, произвольного или связанного с политикой, значения в драйвере.

Остался только вопрос, как были выбраны значения по умолчанию. В данном случае проблема состоит в нахождении лучшего соотношения между потерей памяти в результатеполузаполненного кванта и квантового набора, и накладных расходов на выделения, освобождения, и указатель связывания, что происходит, если кванты и наборы малы. Кроме того, должен быть принят во внимание внутренний дизайн kmalloc. Выбор чисел по умолчанию делается из предположения, что во время тестирования в scull могут быть записаны большие объёмы данных, хотя при обычном использовании устройства, скорее всего, будут передаваться только несколько килобайт данных.

Мы уже видели структуру scull_dev, которая является внутренним представлением нашего устройства. Это поля структуры quantum и qset, содержащие квант и размер квантового набора устройства, соответственно. Фактические данные, однако, отслеживаются другой структурой, которую мы назвали структурой scull_qset:

```
struct scull_qset
{
    void **data;
    struct scull_qset *next;
};
```

Следующий фрагмент кода показывает на практике, как структуры scull_dev и scull_qset используются для хранения данных. Функция scull_trim отвечает за освобождение всей области данных и

вызывается из scull_open, когда файл открывается для записи. Это просто прогулка по списку и освобождение любого кванта и набора квантов при их нахождении.

```
int scull_trim(struct scull_dev *dev)
{
    struct scull_qset *next, *dptr;
    int qset = dev->qset; /* "dev" является не-null */
    int i;
    for (dptr = dev->data; dptr; dptr = next)
    {
        /* все элементы списка */
        if (dptr->data)
        {
            for (i = 0; i < qset; i++)
                kfree(dptr->data[i]);
            kfree(dptr->data);
            dptr->data = NULL;
        }
        next = dptr->next; kfree(dptr);
    }
    dev->size = 0;
    dev->quantum = scull_quantum; dev->qset = scull_qset;
dev->data = NULL;
    return 0;
}
```

scull_trim также используется в функции очистки модуля, чтобы вернуть системе память, используемую scull.

Методы read и write выполняют аналогичные задачи, то есть копирование данных из и в код приложения. Таким образом, их прототипы очень похожи и стоит представить их в одно время:

```
ssize_t read(struct file *filp, char __user *buff, size_t
count, loff_t *offp);
ssize_t write(struct file *filp, const char __user *buff,
size_t count, loff_t *offp);
```

Для обоих методов filp является указателем на file, а count - это размер запрашиваемой передачи данных. Аргумент buff указывает на пользовательский буфер данных, удерживающий данные, которые будут записаны, или пустой буфер, в котором должны быть размещены вновь считанные данные. Наконец, offp является указателем на объект "типом длинного смещения" (“long offset type”), который указывает на позицию файла, к которой обращается пользователь. Возвращаемое значение является "типом знакового размера" (“signed size type”); его использование будет рассмотрено позже.

Давайте повторим, что аргумент buff для методов read и write является указателем пространства пользователя. Поэтому он не может быть непосредственно разыменовываться кодом ядра. Есть несколько причин для этого ограничения:

- В зависимости от архитектуры на которой работает ваш драйвер и как было сконфигурировано ядро, указатель пользовательского пространства может совсем не быть действительным во время работы в режиме ядра. Может не быть ничего связанного с этим адресом, или он может указывать на какие-то другие, случайные данные.
- Даже если указатель означает что-то в пространстве ядра, память пользовательского пространства организована странично и память в запросе может не находиться в ОЗУ, когда сделан этот системный вызов. Попытка сослаться на память пользовательского пространства непосредственно может сгенерировать ошибку страничного доступа, это то, что код ядра не разрешает делать. Результатом будет "Ой" (“oops”), что приведёт к гибели процесса, который сделал этот системный вызов.
- Указатель в запросе поступает от пользовательской программы, которая могла бы быть ошибочной или злонамеренной. Если же ваш драйвер слепо разыменовывает переданный пользователем указатель, он представляет собой открытую дверь, позволяющую программе пользовательского пространства получить доступ или перезаписать память где угодно в системе. Если вы не хотите нести ответственность за ущерб безопасности системам ваших пользователей, вы никогда не можете разыменовывать указатель пользовательского пространства напрямую.

Очевидно, что ваш драйвер должен иметь возможность доступа в буфер пользовательского пространства для того, чтобы выполнить свою работу. Однако, чтобы быть безопасным, этот доступ должен всегда быть выполнен через специальные функции, поставляемые для того ядром. Приведём некоторые из этих функций (которые определены в <asm/uaccess.h>) здесь; они используют некоторую особую, архитектурно-зависимую магию, чтобы гарантировать, что передача данных между ядром и пользовательским пространством происходит безопасным и правильным способом.

Коду для read и write в scull необходимо скопировать весь сегмент данных в или из адресного пространства пользователя. Эта возможность предоставляется следующими функциями ядра, которые копируют произвольный массив байтов и находятся в основе большинства реализаций read и write:

```
unsigned long copy_to_user(void __user *to,
```

```
                const void *from,
                unsigned long count);
unsigned long copy_from_user(void *to,
                const void __user *from,
                unsigned long count);
```

Хотя эти функции ведут себя как нормальные функции memcpy, следует применять немного дополнительной осторожности, когда из кода ядра адресуется пространство пользователя. Адресуемые пользовательские страницы могут не быть в настоящее время в памяти и подсистема виртуальной памяти может отправить процесс в спячку, пока страница не будет передана на место. Это происходит, например, когда страница должна быть получена из свопа. Конечный результат для автора драйвера в том, что любая функция, которая осуществляет доступ к пространству пользователя, должна быть повторно-входимой, должна быть в состоянии выполняться одновременно с другими функциями драйвера, и, в частности, должна быть в состоянии, когда она может законно спать.

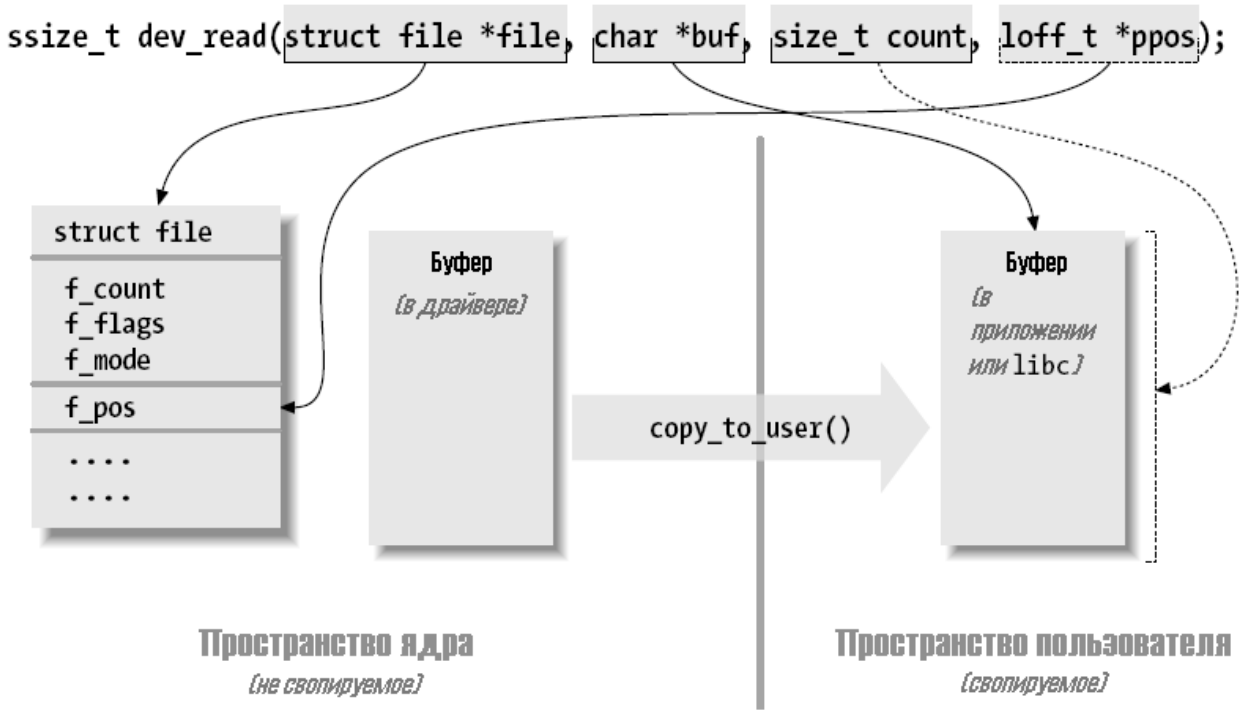
Роль этих двух функций не ограничивается копированием данных в и из пространства пользователя: они также проверяют, является ли указатель пользовательского пространства действительным. Если указатель является недействительным, копирование не выполняется, с другой стороны, если неверный адрес встречается во время копирования, копируется только часть данных. В обоих случаях возвращается значение объёма памяти, которое всё же скопировано. Код scull смотрит на возвращаемую ошибку и возвращает -EFAULT пользователю, если значение не 0.

Тема доступа в пространство пользователя и недействительных указателей пространства пользователя более широка. Однако, стоит заметить, что если вам не требуется проверять указатель пользовательского пространства, вы можете вызвать взамен __copy_to_user и __copy_from_user. Это полезно, например, если вы знаете, что уже проверили аргумент. Тем не менее, будьте осторожны, если в действительности вы не проверяете указатель пользовательского пространства, который вы передаёте в эти функции, вы можете создать аварии ядра и/или дыры в системе безопасности.

Что же касается самих методов устройства, задача метода read - скопировать данные из устройства в пространство пользователя (используя copy_to_user), а метод write должен скопировать данные из пространства пользователя на устройство (с помощью copy_from_user). Каждый системный вызов read или write запрашивает передачу определённого числа байт, но драйвер может свободно передать

меньше данных - точные правила немного отличаются для чтения и записи и описаны далее в этой главе.

Независимо от количества переданных методами данных, обычно они должны обновить позицию файла в *offp, представляющего текущую позицию в файле после успешного завершения системного вызова. Затем когда требуется, ядро передаёт изменение позиции файла обратно в структуру file. Однако, системные вызовы pread и pwrite имеют различную семантику; они работают от заданного смещения файла и не изменяют позицию файла, видимую любым другим системным вызовом. Эти вызовы передают указатель на позицию, заданную пользователем, и отменяют изменения, которые делает ваш драйвер. Следующий рисунок показывает, как использует свои аргументы типичная реализация read.



Оба метода, read и write, если произошла ошибка, возвращают отрицательное значение. Вместо этого, возвращаемое значение большее или равное 0, говорит вызывающей программе, сколько байт было успешно передано. Если какие-то данные передаются правильно и затем происходит ошибка, возвращаемое значение должно быть числом успешно переданных байт и об ошибке не сообщается до следующего вызова функции. Конечно, выполнение этого правила требует, чтобы ваш драйвер помнил о том, что произошла ошибка, чтобы он мог вернуть статус ошибки в будущем.

Хотя функции ядра возвращают отрицательное число как сигнал об ошибке, а значение числа показывает вид произошедшей ошибки, программы, которые выполняются в пользовательском пространстве,

всегда видят -1 в качестве возвращаемого ошибочного значения. Они должны получить доступ к переменной errno, чтобы выяснить, что случилось. Поведение в пользовательском пространстве диктуется стандартом POSIX, но этот стандарт не предъявляет требований к внутренним операциям ядра.

Метод read

Возвращаемое значение для read интерпретируется вызывающей прикладной программой:

- Если значение равно аргументу count, переданному системному вызову read, передано запрашиваемое количество байт. Это оптимальный случай.
- Если число положительное, но меньше, чем count, только часть данных была передана. Это может произойти по ряду причин, в зависимости от устройства. Чаще всего прикладная программы повторяет чтение. Например, если вы читаете с помощью функции fread, библиотечная функция повторяет системный вызов до завершения запрошенной передачи данных.
- Если значение равно 0, был достигнут конец файла (и данные не были прочитаны).
- Отрицательное значение означает, что произошла ошибка. Значение указывает, какая была ошибка согласно <linux/errno.h>. Типичные возвращаемые значения ошибки включают -EINTR (прерванный системный вызов) или -EFAULT (плохой адрес).

Что отсутствует в этом списке, так это случай "нет данных, но они могут прийти позже". В этом случае системный вызов read должен заблокироваться. Мы рассмотрим работу с блокирующим вводом позже.

Код scull использует эти правила. В частности, он пользуется правилом частичного чтения. Каждый вызов scull_read имеет дело только с одним квантом данных, не создавая цикл, чтобы собрать все данные; это делает код короче и проще для чтения. Если читающая программа действительно хочет больше данных, она повторяет вызов. Если для чтения устройства используется стандартная библиотека ввода/вывода (например, fread), приложение даже не заметит квантования при передаче данных.

Если текущая позиция чтения больше размера устройства, метод read драйвера scull возвращает 0, чтобы сигнализировать, что данных больше нет (другими словами, мы в конце файла). Эта ситуация может произойти, если процесс А читает устройство в то время, как процесс Б открывает его для записи, тем самым укорачивая размер устройства

до 0. Процесс вдруг обнаруживает себя в конце файла и следующий вызов read возвращает 0.

Вот код для read (игнорируйте сейчас вызовы down_interruptible и up, мы узнаем о них на последующих занятиях):

```
ssize_t scull_read(struct file *filp, char __user *buf,
size_t count, loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr; /* первый списковый объект
*/
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset;  /* как много байт в
списковом объекте */
    int item, s_pos, q_pos, rest;
    ssize_t retval = 0;
    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    if (*f_pos >= dev->size)
        goto out;
    if (*f_pos + count > dev->size)
        count = dev->size - *f_pos;
/* найти списковый объект, индекс qset, и смещение в
кванте */ item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;
/* следовать за списком до правой позиции (заданной где-
то) */ dptr =        scull_follow(dev, item);
    if (dptr == NULL || !dptr->data || !dptr-
>data[s_pos])
        goto out; /* не заполнять пустые пространства */
/* читать только до конца этого кванта */ if (count >
quantum - q_pos)
    count = quantum - q_pos;
    if (copy_to_user(buf, dptr->data[s_pos] + q_pos,
count))
    {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;
out:
    up(&dev->sem);
    return retval;
}
```

Присвоения данных для 'quantum' и 'qset' (3-я строка) должны быть защищены семафором, чтобы избежать следующих (правда, маловероятных) состояний гонок:

- Данное устройство открыто для чтения каким-либо процессом.
- Кто-то изменяет значение 'quantum' с помощью ioctl().
- Начинается чтение из устройства, читается его значение 'quantum', но семафор еще не захвачен.
- Устройство открывается вторым процессом в режиме O_WRONLY, вызывая его повторную инициализацию, что приводит к изменению значения 'quantum' устройства.
- Этот процесс пишет в устройство scul.
- Возобновление первого чтения

Метод write

write, как и read, может передавать меньше данных, чем было запрошено в соответствии со следующими правилами для возвращаемого значения:

- Если значение равно count, передано запрашиваемое количество байт.
- Если число положительное, но меньше, чем count, была передана только часть данных. Программа, скорее всего, повторит запись остальных данных.
- Если значение равно 0, ничего не записано. Этот результат не является ошибкой и нет никаких оснований для возвращения кода ошибки. И снова стандартная библиотека повторит вызов write. Мы рассмотрим точное значение этого случая в Главе 6 128, где познакомимся с блокирующей записью.
- Отрицательное значение означает ошибку; как и для read, допустимые значения ошибок определены в <linux/errno.h>.

К сожалению, всё равно могут быть неправильные программы, которые выдают сообщение об ошибке и прерываются, когда производится частичная передача. Это происходит потому, что некоторые программисты привыкли к вызовам write, которые либо ошибочны, либо полностью успешны, это происходит в большом количестве случаев и должно так же поддерживаться устройством. Это ограничение в реализации scull может быть исправлено, но мы не хотим усложнять код более, чем необходимо.

Код scull для write выполняется с одним квантом за раз, так же, как это делает метод read:

```
ssize_t scull_write(struct file *filp, const char __user
*buf, size_t count, loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset;
    int item, s_pos, q_pos, rest;
    ssize_t retval = -ENOMEM; /* значение используется в
инструкции "goto out" */
    if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
/* найти списковый объект, индекс qset, и смещение в
кванте */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;
/* следовать за списком до правой позиции */ dptr =
    scull_follow(dev, item);
    if (dptr = = NULL)
        goto out;
    if (!dptr->data)
    {
        dptr->data = kmalloc(qset * sizeof(char *),
GFP_KERNEL);
        if (!dptr->data)
            goto out;
        memset(dptr->data, 0, qset * sizeof(char *));
    }
    if (!dptr->data[s_pos])
    {
        dptr->data[s_pos] = kmalloc(quantum,
GFP_KERNEL);
        if (!dptr->data[s_pos])
            goto out;
    }
/* записать только до конца этого кванта */
    if (count > quantum - q_pos)
        count = quantum - q_pos;
    if (copy_from_user(dptr->data[s_pos]+q_pos, buf,
count))
    {
        retval = -EFAULT;
        goto out;
    }
```

```
    *f_pos += count;
    retval = count;
/* обновить размер */
    if (dev->size < *f_pos)
        dev->size = *f_pos;
out:
    up(&dev->sem);
    return retval;
}
```

Системы Unix уже давно поддерживают два системных вызова, названных readv и writev. Эти "векторные" версии read и write берут массив структур, каждая из которых содержит указатель на буфер и значение длины. Вызов readv будет читать указанное количество в каждый буфер по порядку. writev, вместо этого, будет собирать вместе содержимое каждого буфера и передавать их наружу как одну операцию записи.

Если драйвер не предоставляет методы для обработки векторных операций, readv и writev осуществляются несколькими вызовами ваших методов read и write. Однако во многих случаях путём прямой реализации readv и writev достигается повышение эффективности.

Прототипами векторных операций являются:

```
ssize_t (*readv) (struct file *filp, const struct iovec
*iov, unsigned long count, loff_t *ppos);
ssize_t (*writev) (struct file *filp, const struct iovec
*iov, unsigned long count, loff_t *ppos);
```

Здесь аргументы filp и ppos такие же, как для read и write. Структура iovec, определённая в <linux/uio.h>, выглядит следующим образом:

```
struct iovec
{
    void __user *iov_base;
    __kernel_size_t iov_len;
}
```

Каждая iovec описывает одну порцию данных для передачи; она начинается в iov_base (в пространстве пользователя) и имеет длину iov_len байт. Параметр count говорит методу, сколько существует структур iovec. Эти структуры созданы приложением, но ядро копирует их в пространство ядра до вызова драйвера.

Простейшей реализацией векторных операций будет прямой цикл, который просто передаёт адрес и длину каждой iovec функциям драйвера read или write. Часто, однако, эффективное и правильное поведение требует, чтобы драйвер делал что-то поумнее. Например,

writev на ленточном накопителе должна сохранить содержимое всех структур iovec одной операцией записи на магнитную ленту.

Однако, многие драйверы не получили никакой выгоды от реализации самими этих методов. Таким образом, scull опускает их. Ядро эмулирует их с помощью read и write, и конечный результат тот же.

Как только вы оснастили драйвер четырьмя вышеописанными методами, драйвер может быть собран и протестирован; он сохраняет любые данные, которые вы запишете в него, пока вы не перезапишете их новыми данными. Устройство действует как буфер данных, размер которого ограничен только реально доступной памятью. Для проверки драйвера вы можете попробовать использовать cp, dd и перенаправление ввода/вывода.

Чтобы увидеть, как сжимается и расширяется объём свободной памяти в зависимости от того, как много данных записано в scull, может быть использована команда free.

Чтобы стать более уверенными при чтении и записи одного кванта за раз, вы можете добавить printk в соответствующую точку в драйвере и посмотреть, что происходит в то время, как приложение читает или записывает большие массивы данных. Альтернативно, используйте утилиту strace для мониторинга системных вызовов вместе с их возвращаемыми значениями, выполняемыми программой. Трассировка cp или ls -l > /dev/scull0 показывает квантованные чтения и записи.