Создание класса GEOM

Аннотация

Эта статья документирует некоторые начальные выкладки в разработке GEOM-классов, а также модулей ядра в общем. Предполагается, что читатель близко знаком с программированием на Си в контексте пространства пользовательских процессов (userland).

Содержание

1. Введение 1
2. Подготовка 2
3. Программирование в ядре FreeBSD 4
4. Программирование в системе GEOM 6

1. Введение

1.1. Documentation

Документация по программированию для ядра скудная, это одна из немногих областей программирования, где почти нет хороших учебных пособий, и совет "читай исходники!" - сохраняет свою справедливость. Однако, существует несколько статей и книг разной актуальности, которые рекомендуются к изучению перед тем, как начать программировать:

- Руководство FreeBSD для разработчиков - часть Проекта Документации FreeBSD, ничего специфичного о программировании ядра в нём нет, зато есть немного общей полезной информации.
- Руководство по Архитектуре FreeBSD - также является частью Проекта Документации FreeBSD, содержит описания некоторых низкоуровневых средств и процедур. Уделите внимание разделу номер 13 - Написание драйверов устройств для FreeBSD.
- Несколько интересных статей об устройстве ядра можно найти на сайте FreeBSD Diary.
- Страницы из раздела номер 9 системного справочника, содержат важную документацию по функциям ядра.
- Страница справочника geom(4), а также слайды Пола-Хеннинга Кампа - общее представление о подсистеме GEOM.
- Страницы справочника g_bio(9), g_event(9), g_data(9), g_geom(9), g_provider(9), g_consumer(9), g_access(9), а также другие, связанные с вышеупомянутыми и раскрывающие специфический функционал подсистемы GEOM.

- Страница справочника style(9) - документирует соглашения о стиле оформления кода, которые обязаны быть соблюдены если вы планируете передать ваш код в Subversion репозиторий FreeBSD.

2. Подготовка

Для того, чтоб заниматься разработками для ядра, желательно иметь два отдельных компьютера. Один из них предназначен для среды разработки и исходных кодов, а второй - для запуска тестов отлаживаемого кода. Второму компьютеру для работы достаточно иметь возможность выполнять начальную загрузку по сети и монтирование файловых систем по сети. В этой ситуации, если отлаживаемый код содержит ошибки и вызовет аварийную остановку системы, то это не повлечет порчу или утерю исходного кода . Второму компьютеру даже не потребуется иметь свой монитор, достаточно будет соединения асинхронных портов кабелем RS-232 или соединения при помощи KVM-устройства.

Но так как далеко не у каждого есть два или более компьютеров под рукой, есть пара способов подготовить иную "живую" систему для разработки кода для ядра. Один из них - это разработка в VMWare или QEmu виртуальной машине (это лучшее из доступного, после, конечно-же, выделенного для тестов компьютера).

2.1. Настройка системы для разработки

Прежде всего необходимо иметь в ядре поддержку `INVARIANTS`. Добавьте следующие строки в файл конфигурации ядра:

```
options INVARIANT_SUPPORT
options INVARIANTS
```

Для большей информативности при отладке включите поддержку WITNESS, которая будет предупреждать вас в случае возникновения взаимоблокировок:

```
options WITNESS_SUPPORT
options WITNESS
```

Также включите отладочные символы, если планируете выполнять отладку по дампам аварийных отказов:

```
  makeoptions    DEBUG=-g
```

Установка отладочного ядра обычным способом (`make installkernel`) не даст привычного результата: файл ядра будет называться kernel.debug и будет находиться в /usr/obj/usr/src/sys/KERNELNAME/. Для удобства, отладочное ядро необходимо скопировать в /boot/kernel/.

Также удобно иметь включённый отладчик ядра, так вы сможете исследовать паники

сразу-же после их возникновения. Для включения отладчика добавьте следующие строки в файл конфигурации ядра:

```
options KDB
options DDB
options KDB_TRACE
```

Для автоматического запуска отладчика ядра после возникновения паники может понадобиться установить переменную sysctl:

```
  debug.debugger_on_panic=1
```

Паники системы будут происходить, поэтому уделите внимание кэшу файловой системы. Обычно, при включённом механизме softupdates, последняя версия файла может быть утеряна если паника произошла раньше сбрасывания кэша на устройство хранения. Выключение механизма softupdates (посредством монтирования файловой системы с опцией "sync") значительно сказывается на производительности и, опять-же, не гарантирует целостности данных. Как компромисс, можно сократить задержки сбрасывания кэша механизма softupdates. Есть три переменных sysctl, значения которых необходимо изменить (лучше всего - прописав их в /etc/sysctl.conf):

```
kern.filedelay=5
kern.dirdelay=4
kern.metadelay=3
```

Значения этих переменных - секунды.

Для отладки паник ядра необходимы дампы памяти. Так как паника ядра может "сломать" файловую систему, дамп сначала сохраняется в "сырой" раздел. Обычно, это своп-раздел. Поэтому, размер своп-раздела должен быть не меньше размера ОЗУ компьютера. При последующей загрузке дамп копируется в обычный файл. Это происходит сразу-же после проверки и монтирования файловых систем, но перед активированием раздела свопа. Такое поведение контролируется следующими переменными /etc/rc.conf:

```
dumpdev="/dev/ad0s4b"
dumpdir="/usr/core"
```

Переменная `dumpdev` указывает на раздел подкачки, а `dumpdir` сообщает системе куда перемещать дамп ядра при следующей загрузке.

Сохранение дампа ядра - процесс медленный, и, если у вашего компьютера много оперативной памяти (>256M) и если паники случаются часто, то ожидание сохранения дампов может начать раздражать (вспомним, что над дампом происходит две операции: сохранение в своп-файл и перемещение на файловую систему). В таком случае может оказаться удобным ограничивание объёма используемой системой памяти путем

установки переменной в /boot/loader.conf:

```
hw.physmem="256M"
```

Если паники случаются часто и размер файловых систем большой (или же вы просто не доверяете softupdates и фоновой проверке файловых систем), рекомендуется отключить фоновую проверку файловых систем посредством установки переменной в /etc/rc.conf:

```
background_fsck="NO"
```

В этом случае файловые системы будут проверяться только при необходимости. Также заметьте, что в случае использования фоновой проверки, новая паника может случиться в то время, когда проверяются диски. Другими словами, наиболее безопасный способ - не иметь много локальных файловых систем, а использовать второй компьютер в качестве NFS-сервера.

2.2. Начало проекта

Для написания нового класса GEOM необходимо создать подкаталог в любом доступном пользователю каталоге. Совсем не обязательно, чтоб ваш модуль изначально размещался в /usr/src.

2.3. Makefile

Правилом хорошего тона является создание Makefile-ов для каждого нетривиального проекта, примером которого конечно-же является создание модулей ядра.

Создание Makefile - дело не сложное благодаря исчерпывающему набору вспомогательных средств, предоставляемых системой. В вкратце, вот как должен выглядеть Makefile для модуля ядра:

```
SRCS=g_journal.c
KMOD=geom_journal

.include <bsd.kmod.mk>
```

Этот Makefile (с изменёнными именами файлов) подойдет к любому модулю ядра. Класс GEOM может размещаться в одном единственном модуле ядра. Если для сборки вашего модуля требуется больше, чем один файл, то перечислите их имена, разделённые пробельными символами, в переменной `SRCS`.

3. Программирование в ядре FreeBSD

3.1. Выделение памяти

Прочитайте malloc(9) - выделение памяти лишь немного отличается от своего эквивалента, используемого в пространстве пользовательских процессов (userland). Наиболее приметно то, что `malloc`() и `free`() принимают дополнительные параметры, которые описаны в странице справочника.

Тип "malloc_type" необходимо объявить в секции деклараций файла с исходным кодом, например:

```
static MALLOC_DEFINE(M_GJOURNAL, "gjournal data", "GEOM_JOURNAL Data");
```

Для того, чтобы можно было использовать этот макрос, необходимо включить следующие заголовочные файлы: sys/param.h, sys/kernel.h и sys/malloc.h.

Существует ещё один механизм выделения памяти - UMA (Universal Memory Allocator), описанный в uma(9). Это специфический метод, преимущественно предназначенный для быстрого выделения памяти под списки, состоящие из элементов одинакового размера (например, динамические массивы структур).

3.2. Очереди и списки

Ознакомьтесь с queue(3) Во множестве случаев вам необходимо будет организовывать и управлять такой структурой данных, как списки. К счастью, эта структура данных реализована несколькими способами в виде макросов на Си, а также включена в систему. Наиболее гибкий и часто употребляемый тип списка - TAILQ. Этот тип списка также один из наиболее требовательных к памяти (его элементы - с двойными связями), а также - наиболее медленный (однако счет идёт на несколько инструкций ЦПУ, поэтому последнее утверждение не следует воспринимать в всерьез).

Если важна скорость получения данных, то возьмите на вооружение tree(3) и hashinit(9).

3.3. BIOs

Структура `bio` используется для всех операций ввода/вывода, касающихся GEOM. Она содержит информацию о том, какое устройство ('поставщик geom') должно ответить на запрос, тип запроса, смещение, длину и указатель на буфер, а также набор "определённых пользователем" флагов и полей .

Важным моментом является то, что `bio` обрабатываются асинхронно. Это значит, что во многих частях кода нет аналога к read(2) и write(2) функциям из пространства пользовательских процессов, которые не возвращают управление пока не выполнится системный вызов. Скорее, по завершении обработки запроса (или в случае ошибки при обработке) как извещение вызывается определённая пользователем функция.

Асинхронная модель программирования в чем-то сложней, нежели чаще используемая императивная модель, используемая в пространстве пользовательских процессов; в любом

случае, привыкание займёт некоторое время. В некоторых случаях могут быть использованы вспомогательные функции `g_write_data`() и `g_read_data`(), но *далеко не всегда*. В частности, эти функции не могут использоваться когда захвачен мьютекс; например, мьютекс GEOM-топологии или внутренний мьютекс, удерживаемый в ходе выполнения `.start`() или `.stop`().

4. Программирование в системе GEOM

4.1. Ggate

Если максимальная производительность не требуется, то более простой способ совершать преобразования данных - это выполнять их в пространстве пользовательских процессов посредством ggate (GEOM gate). К недостаткам следует отнести невозможность простого переноса кода в ядро.

4.2. Класс GEOM

Класс GEOM выполняет преобразования данных. Эти преобразования могут быть скомпонованы друг с другом в виде дерева. Экземпляр класса GEOM называют *geom*.

В каждом классе GEOM есть несколько "методов класса", которые вызываются когда экземпляра класса нет в наличии (или же они не привязаны к конкретному экземпляру класса):

- `.init` вызывается тогда, когда системе GEOM становится известно о классе GEOM (например, когда загружается модуль ядра).
- `.fini` будет вызван в случае отказа GEOM системы от класса (например, при выгрузке модуля)
- `.taste` вызывается, когда в системе появляется новый класс или поставщик geom ("provider"). Если соответствие найдено, то эта функция обычно создаёт и запускает экземпляр geom.
- `.destroy_geom` вызывается при необходимости разрушить экземпляр geom
- `.ctlconf` будет вызван, когда пользователь запросит изменение конфигурации существующего экземпляра geom

Также определены функции событий GEOM, которые копируются в экземпляр geom.

Поле `.geom` в структуре `g_class` - это список (LIST) экземпляров geom, реализованных из класса.

Эти функции вызываются из g_event потока ядра.

4.3. Softc

"softc" - это устаревший термин для "приватных данных драйвера" ("driver private data").

Название вероятней всего происходит от устаревшего термина "software control block". В системе GEOM softc это структура (точнее: указатель на структуру) которая может быть присоединена к экземпляру geom и может содержать приватные данные экземпляра. У большинства классов GEOM есть следующие члены:

- `struct g_provider *provider` : "поставщик geom" предоставляемый данным экземпляром geom
- `uint16_t n_disks` : Количество потребителей geom ("consumer"), обслуживаемых данным экземпляром geom
- `struct g_consumer **disks` : Массив `struct g_consumer*`. (Невозможно обойтись одинарным указателем, потому что система GEOM создаёт для нас структуры struct g_consumer).

Структура `softc` содержит состояние экземпляра geom. У каждого экземпляра есть свой softc.

4.4. Метаданные

Формат метаданных в той или иной мере зависит от конкретного класса, но *обязан* начинаться с:

- 16-байтного буфера для подписи - строки с завершающим нулем (обычно это имя класса)
- uint32 идентификатора версии

Подразумевается, что классы geom знают как обращаться с метаданными с идентификаторами версий ниже, чем их собственные.

Метаданные размещаются в последнем секторе поставщика geom (поэтому обязаны целиком умещаться в нем).

(Все это зависит от реализации, но весь существующий код работает подобно описанному и поддерживается библиотеками.)

4.5. Маркирование/создание экземпляра geom

Последовательность событий следующая:

- пользователь запускает служебную программу geom(8)
- программа решает каким классом geom ей придется управлять и ищет библиотеку geom_CLASSNAME.so (которая обычно находится в /lib/geom).
- она открывает библиотеку при помощи dlopen(3), извлекает вспомогательные функции и определения параметров командной строки.

Вот так происходит создание/маркирование нового экземпляра geom:

- geom(8) ищет команду в аргументах командной строки (обычно это `label`) и вызывает вспомогательную функцию.
- Вспомогательная функция проверяет параметры и собирает метаданные, которые записываются во все вовлеченные поставщики geom.

- Это "повреждает (spoil)" существующие экземпляры geom (если они были) и порождает новый виток "тестирования" поставщиков geom. Целевой класс geom опознает метаданные и активирует экземпляр geom.

(Приведенная выше последовательность событий зависит от конкретной реализации, но весь существующий код работает подобно описанному и поддерживается библиотеками.)

4.6. Структура команд geom

Вспомогательная библиотека geom_CLASSNAME.so экспортирует структуру `class_commands`, которая является массивом элементов `struct g_command`. Эти команды одинакового формата и выглядят следующим образом:

```
verb [-options] geomname [other]
```

Общими командами являются:

- label - записать метаданные в устройства, чтобы они могли быть опознаны в процессе тестирования и использованы в соответствующих экземплярах geom
- destroy - разрушить метаданные, за которым последует разрушение экземпляров geom

Общие опции:

- `-v` : детальный вывод
- `-f` : принудить

Некоторые операции, к примеру маркирование метаданными и разрушение метаданных могут быть выполнены из пространства пользовательских процессов. Для этого, структура `g_command` содержит поле `gc_func`, которое может быть установлено на функцию (в том-же .so), которая будет вызвана для обработки команды. В случае, когда `gc_func` равно NULL, команда будет передана модулю ядра: функции `.ctlreq` класса GEOM.

4.7. Экземпляры geom

У экземпляров классов GEOM есть внутренние данные, которые хранятся в структурах softc, а также есть некоторые функции, посредством которых они реагируют на внешние события.

Функции событий:

- `.access` : просчитывает права доступа (чтение/запись/исключительный доступ)
- `.dumpconf` : возвращает информацию о экземпляре geom; формат XML
- `.orphan` : вызывается, когда отсоединяется любой из низлежащих поставщиков geom
- `.spoiled` : вызывается, когда производится запись в низлежащий поставщик geom
- `.start` : обрабатывает ввод/вывод

Эти функции вызываются из ядерного потока `g_down` и в этом контексте не может быть блокировок (поищите определение "блокировка" в других источниках), что немного ограничивает свободу действий, но способствует быстроте обработки.

Из вышеупомянутых, наиболее важной и выполняющей полезную работу функцией является `.start`(), которая вызывается всякий раз, когда поставщику geom, управляемому экземпляром класса, приходит запрос BIO.

4.8. Потоки выполнения системы geom

Системой GEOM в ядре ОС создаются и используются три потока выполнения (kernel threads):

- `g_down` : Обрабатывает запросы, приходящие от высокоуровневых сущностей (таких как запросы из пространства пользовательских процессов) на пути к физическим устройствам
- `g_up` : Обрабатывает ответы от драйверов устройств на запросы, выполненные высокоуровневыми сущностями
- `g_event` : Отрабатывает в остальных случаях, как-то создание экземпляра geom, просчитывание прав доступа, события "повреждения" и т.п.

Когда пользовательский процесс запрашивает "прочитать данные X по смещению Y файла", происходит следующее:

- Файловая система преобразует запрос в экземпляр структуры bio и передаёт его системе GEOM. Файловая система "знает", что экземпляр geom должен обработать запрос, так как файловые системы размещаются непосредственно над экземпляром geom.
- Запрос завершается вызовом функции `.start`() в потоке g_down и достигает верхнего экземпляра geom.
- Верхний экземпляр geom (например, это секционировщик разделов (partition slicer)) определяет, что запрос должен быть переадресован нижестоящему экземпляру geom (к примеру, драйверу диска). Вышестоящий экземпляр geom создаёт копию запроса bio (запросы bio *ВСЕГДА* копируются при передаче между экземплярами geom при помощи `g_clone_bio`()!), изменяет поля смещения и целевого поставщика geom и запускает на обработку копию при помощи функции `g_io_request`()
- Драйвер диска также получает запрос bio, как вызов функции `.start`() в потоке `g_down`. Драйвер обращается к контроллеру диска, получает блок данных и вызывает функцию `g_io_deliver`() используя копию запроса bio.
- Теперь, извещение о завершении bio "всплывает" в потоке `g_up`. Сначала в потоке `g_up` вызывается функция `.done`() секционировщика разделов, последний использует полученную информацию, разрушает клонированный экземпляр структуры bio посредством `g_destroy_bio`() и вызывает `g_io_deliver`() используя первоначальный запрос.
- Файловая система получает данные и передаёт их пользовательскому процессу.

За информацией о том, как данные передаются в структуре `bio` между экземплярами geom, смотрите g_bio(9) (обратите внимание на использование полей `bio_parent` и `bio_children`).

Важный момент в том, что *НЕЛЬЗЯ ДОПУСКАТЬ БЛОКИРОВОК В ПОТОКАХ G_UP И G_DOWN*. Вот неполный перечень того, что нельзя делать в этих потоках:

- Вызывать функции `msleep`() или `tsleep`().
- Использовать функции `g_write_data`() и `g_read_data`(), так как они блокируются в момент обмена данными с потребителями geom.
- Ожидать ввод/вывод.
- Вызывать malloc(9) и `uma_zalloc`() с установленным флагом `M_WAITOK`
- Использовать sx(9)

Это ограничение на код GEOM призвано избежать от "засорения" пути запроса ввода/вывода, так как блокировки обычно не имеют чётких временных границ, и нет гарантий на занимаемое время (также на то есть и другие технические причины). Это также значит, что в вышеупомянутых потоках сколь-нибудь сложные операции выполнить нельзя, например: любое сложное преобразование требует выделения памяти. К счастью решение есть: создание дополнительных ядерных потоков.

4.9. Ядерные потоки выполнения, предназначенные для использования в коде geom

Ядерные потоки выполнения создаются функцией kthread_create(9), в своём поведении они схожи с потоками, созданными в пространстве пользовательских процессов, но есть одно отличие: они не могут известить вызвавший их поток о своём завершении; по завершению - необходимо вызывать kthread_exit(9).

В коде GEOM обычное назначение этих потоков - разгрузить поток `g_down` (функцию `.start`()) от обработки запросов. Эти потоки подобны "обработчикам событий" ("event handlers"): у них есть очередь событий (которая наполняется событиями от разных функций из разных потоков; очередь необходимо защищать мьютексом), события из очереди выбираются одно за другим и обрабатываются в большом блоке `switch`().

Основное преимущество использования отдельного потока, который обрабатывает запросы ввода/вывода, то, что он может блокироваться по мере необходимости. Это, несомненно, привлекательно, но должно быть хорошо обдумано. Блокирование - хорошо и удобно, но может существенно снизить производительность преобразований данных в системе GEOM. Особо требовательные к производительности классы могут делать всю работу в функции `.start`(), уделяя особое внимание ошибкам при работе с памятью.

Еще одно преимущество потока "обработчика событий" это сериализация всех запросов и ответов, приходящих с разных потоков geom в один поток. Это также удобно, но может быть медленным. В большинстве случаев, обработка запросов функцией `.done`() может быть оставлена потоку `g_up`.

У мьютексов в ядре FreeBSD (mutex(9)) есть одно различие с их аналогами из пространства пользовательских процессов - во время удержания мьютекса в коде не должно быть блокировки. Если в коде необходимо блокирование, то лучше использовать sx(9). С другой стороны, если вся ваша работа выполняется в одном потоке, вы можете обойтись вообще

без мьютексов.