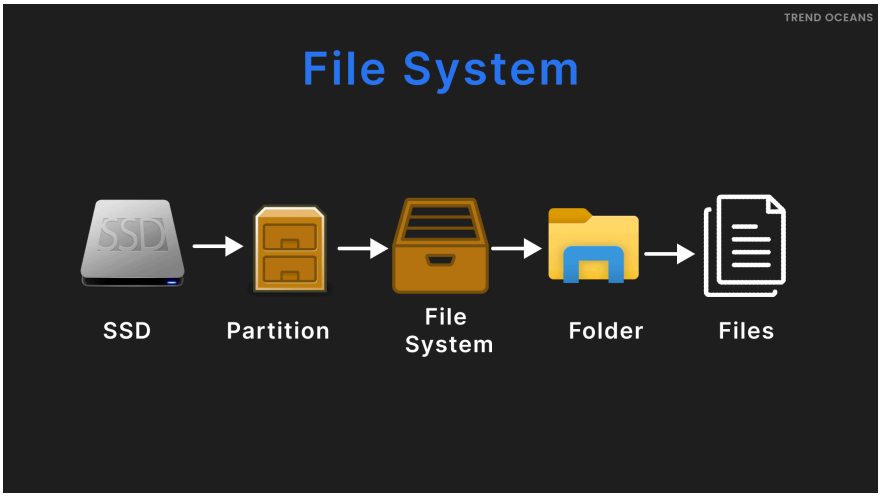
FUSE: как написать свою файловую систему

Меня зовут Максим, я ведущий разработчик в VK. Занимаюсь инфраструктурой доставки электронной почты в проекте Mail.ru. Наша команда разработала и довела до эксплуатации файловую систему (ФС) на FUSE в рамках проекта распределённой почтовой очереди. В проекте требовалось реализовать сетевую ФС, которая сохраняет данные в трёх копиях, в разных ЦОДах. Цель — повысить отказоустойчивость, чтобы полный выход из строя одного ЦОД не приводил к нарушениям SLA. Эта статья для всех, кто интересуется файловыми системами и хранением данных. Мы обсудим:

- зачем писать свою ФС;
- как написать свою ФС с помощью фреймворка FUSE;
- какие подводные камни есть у эксплуатации FUSE в продакшн.

Если слова *dentry*, *inode* и ***монтирование*** вам не знакомы, можете читать статью от начала до конца, это будет полезно для кругозора. Если вы уже знакомы с FUSE и вас интересует исключительно практический опыт, то пропустите теорию и сразу переходите к разделу «II. Подводные камни разработки и эксплуатации FUSE-файловой системы».

Эта статья — результат трёх лет разработки ФС. Сейчас самое время заварить чай, рассказ будет долгим.



I. Как разработать файловую систему с помощью фреймворка FUSE

В Linux одновременно могут работать несколько файловых систем. Каждая из них отвечает за своё поддерево файлов. Корень этого поддерева — директория, которую называют **точка монтирования**, или **mount point**. Чтобы посмотреть список примонтированных ФС в Linux, выполните команду findmnt.

Чтобы запустить новую ФС, необходимо **примонтировать** её с помощью системного вызова mount. В его аргументах указывают, какую ФС, по какому пути и с какими опциями примонтировать. Mount — привилегированный системный вызов. Он влияет на то, что видят другие пользователи в дереве файлов. Поэтому для выполнения `mount` требуется CAP_SYS_ADMIN — фактически, права супер-пользователя.

Если вы решили написать свою файловую систему, то есть два пути: разработать модуль ядра ОС или написать приложение на фреймворке FUSE (Filesystem in Userspace). У второго варианта есть несколько преимуществ:

1. Файловые системы можно разрабатывать как user space-программы, без необходимости писать код в ядре ОС.
2. FUSE поддерживает непривилегированное монтирование ФС, права супер-пользователя не нужны для запуска файловой системы.

Это значит, что на FUSE быстрее разрабатывать и проще отлаживать. Рассмотрим фреймворк подробнее.

Знакомство с FUSE

Файловая система с точки зрения FUSE — это daemon в user space, его называют **драйвером ФС**. Файловые системы в Linux реализуют одинаковый интерфейс. За это отвечает vfs — подсистема ядра Linux. Vfs выполняет роль маршрутизатора: направляет запросы клиентов к экземплярам ФС.

FUSE-модуль ядра:

1. получает запрос от vfs;
2. направляет его драйверу ФС;
3. ждёт от драйвера ответа;
4. возвращает результат обратно в vfs.

Этот процесс показан на рисунке.

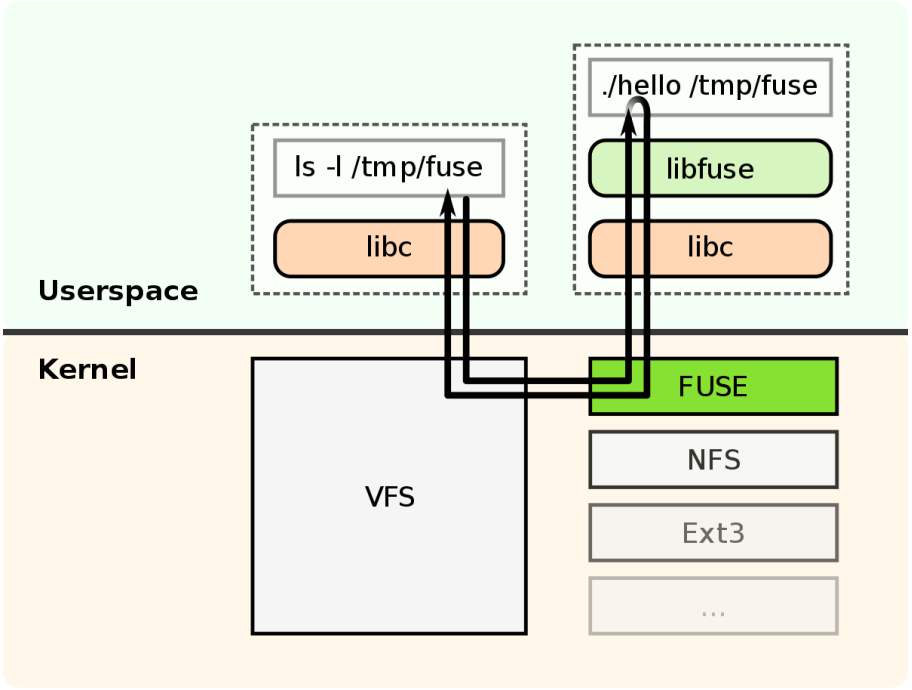


Схема работы фреймворка FUSE

FUSE-модуль ядра ОС и драйвер ФС общаются по RPC-протоколу. Драйвер устанавливает соединение с FUSE-модулем ядра ОС в процессе монтирования, соединения устанавливается через устройство /dev/fuse. По FUSE-соединению ходят пакеты в режиме запрос-ответ. Порядок ответов драйвера ФС может отличаться от порядка, в котором были отправлены запросы. Каждый ответ ссылается на запрос по ID, что позволяет драйверу ФС обрабатывать запросы параллельно.

Далее мы рассмотрим архитектуру драйвера файловой системы.

Как устроен драйвер файловой системы

Драйвер ФС — это RPC-сервер, который обслуживает запросы от FUSE-модуля ядра ОС. Протокол общения реализует библиотека libfuse, она отвечает за жизненный цикл FUSE-соединения, парсинг запросов и сериализацию ответов.

У libfuse есть высокоуровневый и низкоуровневый API. Первый — это обёртка над вторым, он упрощает интерфейс взаимодействия, но снижает производительность. А второй напрямую работает с запросами от ядра ОС. Ключевое отличие: в высокоуровневом API ФС работает с файлами по путям, а в низкоуровневом — напрямую с inode и dentry, подробнее об этом поговорим чуть дальше.

Я пишу про низкоуровневый API, потому что он даёт бóльшую гибкость и позволяет лучше понять устройство файловых систем. Код, на который я ссылаюсь, написан на C. Впрочем, есть биндинги libfuse к языкам высокого уровня и альтернативные реализации библиотеки. Возможно, писать ФС на языке высокого уровня в вашем случае будет удобнее.

Сессия libfuse

Разработка драйвера ФС начинается с создания сессии libfuse, для этого надо вызвать функцию fuse_session_new(). Эта функция принимает на вход структуру struct fuse_lowlevel_ops. Заполненная структура — это RPC API вашей ФС, в ней размещаются указатели на функции — обработчики запросов. Не обязательно реализовывать все обработчики, можно начать с небольшого набора. Клиенты получат ошибку от FUSE, если их запрос не поддерживается.

Созданная сессия libfuse монтируется в директорию с помощью функции fuse_session_mount(). Эта функция реализует под капотом «магию» непривилегированного монтирования. Если привилегий на сисколл mount не хватает, она вызовет программу fusermount. Эта программа поставляется вместе с libfuse. Она выполняется с правами root-пользователя благодаря SUID-биту в правах доступа.

Цикл обработки событий запускается после примонтирования сессии libfuse. В этом цикле драйвер ФС читает запросы из FUSE-соединения и выполняет обработчики. Запустить цикл обработки событий можно несколькими способами.

- **Цикл обработки событий по умолчанию.** Функция fuse_session_loop() обслуживает FUSE-соединение в текущем потоке. Она возвращает управление после того, как ФС будет отмонтирована. Это самый простой вариант цикла обработки событий.
- **Многопоточная обработка событий.** Функция fuse_session_loop_mt() запускает пул потоков. Входящий запрос обрабатывается любым свободным потоком. Обратите внимание, что многопоточная обработка событий создаёт условия для гонок. Используя этот подход, придётся обкладывать структуры данных мьютексами.
- **Самописный цикл обработки событий.** Libfuse предоставляет API для своих реализаций цикла. Функция fuse_session_fd() возвращает файловый дескриптор FUSE-соединения. Функция fuse_session_receive_buf() читает запрос из этого дескриптора в буфер. Функция fuse_session_process_buf() парсит запрос из буфера и выполняет обработчик, указанный в struct fuse_lowlevel_ops при создании сессии libfuse. Функция fuse_session_exited() даёт условие выхода из цикла.

В самом простом варианте, разработка ФС начинается с примерно такого кода:

```C/C++
struct fuse_args args = FUSE_ARGS_INIT(0, NULL);
fuse_opt_add_arg(&args, "");
fuse_opt_add_arg(&args, "-odefault_permissions");
fuse_opt_add_arg(&args, "-oauto_unmount");
fuse_opt_add_arg(&args, "-odebug");

struct fuse_conn_info_opts *opts = fuse_parse_conn_info_opts(&args);
if (!opts)
```

```
    exit(EXIT_FAILURE);

struct fuse_lowlevel_ops api = {
    // TODO: add filesystem handlers
};

struct fuse_session *se = fuse_session_new(&args, &api, sizeof(api), NULL);
if (!se)
    exit(EXIT_FAILURE);

int err = fuse_session_mount(se, "/mnt/my-fs");
if (err)
    exit(EXIT_FAILURE);

fuse_session_loop(se);
```

В проекте мы выбрали самописную реализацию цикла обработки событий. Подробнее об этом поговорим в разделе «Модель многопоточности».

Далее рассмотрим, как написать обработчики запросов.

Структуры данных файловой системы

API FUSE строится вокруг трёх основных структур данных. Рассмотрим эти структуры подробнее.

Inode. Блок произвольных данных с целочисленным ID. В нём хранится содержимое файлов. Также содержит атрибуты из структуры struct stat. Обратите внимание на поле `st_mode`: кроме прав доступа это поле содержит тип файла, например: регулярный файл, директория, символическая ссылка. Для разных типов файлов содержимое inode можно интерпретировать по-разному. Inode регулярных файлов содержат данные; inode директорий ничего не содержат, файлы включаются в директорию с помощью dentry; inode символических ссылок содержат путь, на который ссылаются. Это общая идея, а реализация логики может отличаться в разных ФС.

Dentry. Положение inode в дереве файлов. Содержит имя файла, идентификаторы inode и родительской inode (директории). При монтировании ФС, для корневой директории необходимо создать inode с ID, равным 1. Это well-known значение, все dentry в ФС будут потомками этой inode. Разделение файлов на две независимых сущности dentry и inode — это оптимизация. Она позволяет дёшево переименовывать файлы и перемещать inode между директориями. При этом меняются только dentry, а inode остаются неизменными, копирования данных не происходит.

File. Контекст открытого файла. Когда клиент открывает файл, ФС может создать произвольную структуру и разместить указатель на неё в `fi->fh`. FUSE передаёт в ФС

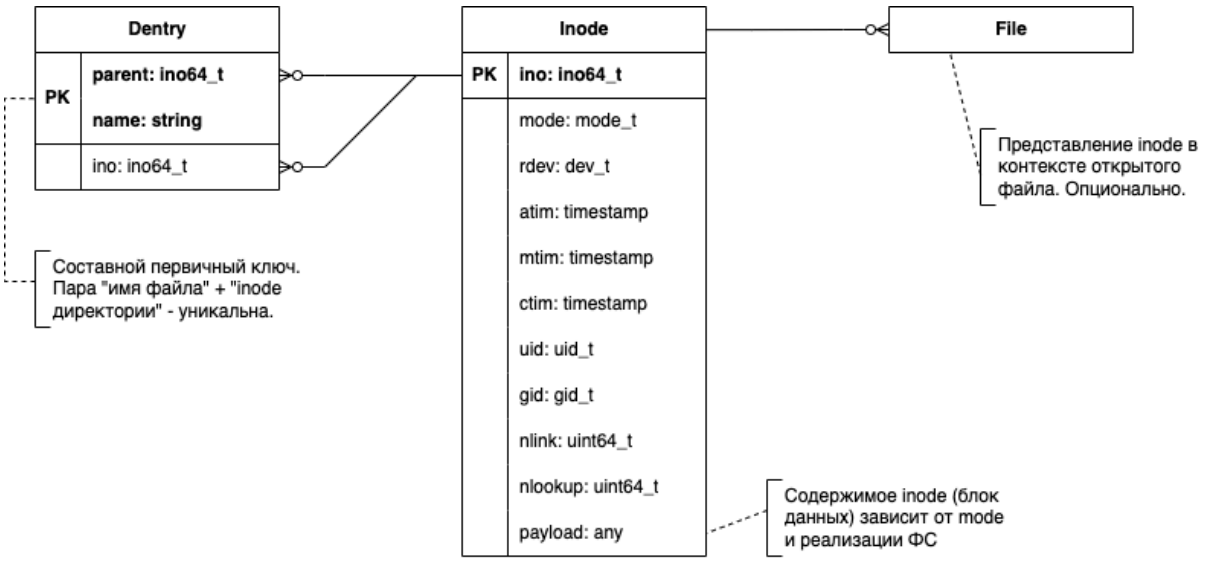
структуру `fi` при каждой операции с открытым файлом. Поле `fi->fh` необязательное, его можно не заполнять. В этом случае ФС реализует **Stateless I/O**, она находит inode по ID при каждой операции с открытым файлом, структура `file` в ФС не объявляется. Если ФС размещает что-то в `fi->fh`, то она реализует **Stateful I/O** — это оптимизация, она позволяет не искать inode по ID при каждой операции с открытым файлом. Это экономит ресурсы процессора и исключает ошибки при поиске.

Структура file вводит в заблуждение. Во-первых, это не «именованная последовательность байтов», как мы привыкли из определений. Во-вторых, в руководствах встречается два словосочетания:

- **file descriptor** — неотрицательное число, ID открытого файла, который программа получает при вызове open();
- **file description** — объект ядра, контекст открытого файла, про него тоже написано в руководстве к open().

File — это file description. Файловые дескрипторы ссылаются на file (description) по ID в таблице открытых файлов процесса. При этом несколько файловых дескрипторов могут ссылаться на один file description.

Разработку своей ФС начинайте с inode и dentry, без структуры file. Этот компромисс позволит сэкономить время и избежать масштабных изменений в архитектуре в будущем. Переход от stateless к stateful I/O простой, если сравнивать с разделением именованных файлов на inode и dentry. Далее рассмотрим жизненный цикл описанных сущностей: как они создаются и удаляются.



Модель данных файловой системы

Жизненный цикл структур данных файловой системы

Жизненный цикл inode. ФС создаёт inode, когда пользователь вызывает mknod(), mkdir() или symlink(). У inode есть счётчик ссылок — поле `nlookup`. В момент создания inode `nlookup = 1`. Linux увеличивает `nlookup`:

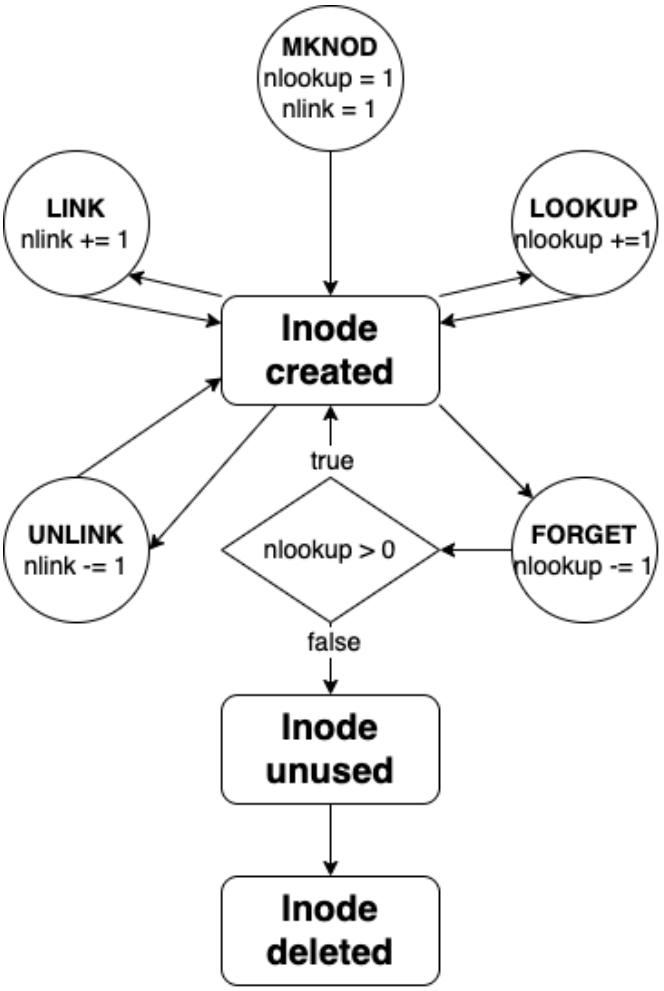
- при создании жёстких ссылок на inode;
- при открытии файла;
- при дублировании открытого файлового дескриптора, в том числе при создании дочернего процесса.

OS вызывает lookup(), чтобы увеличить `nlookup`. Пока на inode ссылаются жёсткие ссылки или файловые дескрипторы, её `nlookup > 0`. Linux уменьшает `nlookup`:

- при закрытии файлового дескриптора;
- при удалении жёсткой ссылки.

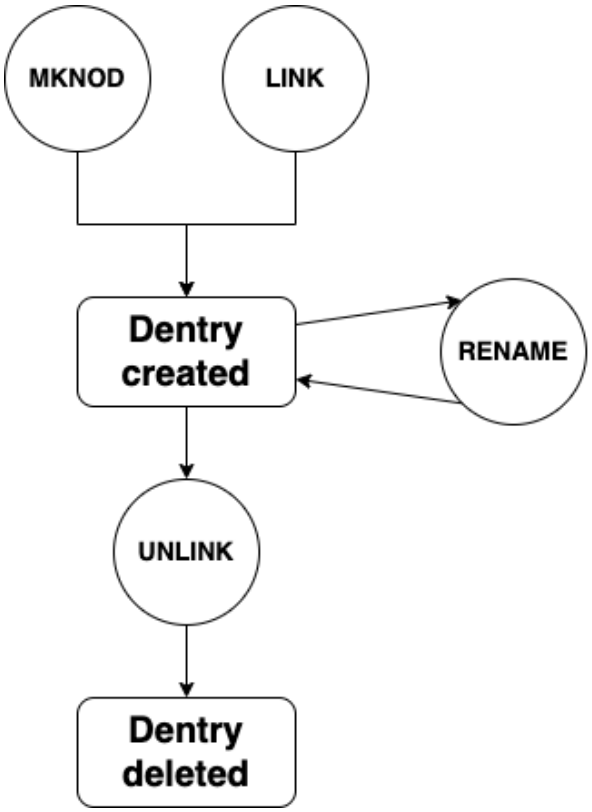
В обоих случаях Linux вызывает forget(), чтобы уменьшить `nlookup`. ФС удаляет inode, когда `nlookup` опускается до нуля.

У inode есть второй счётчик ссылок — `nlink`. Он отражает количество dentry, которые ссылаются на inode — то есть количество жёстких ссылок. Обычно, `nlink <= nlookup`, потому что `nlookup` учитывает открытые файлы и жёсткие ссылки, а `nlink` — только жёсткие ссылки. Но есть исключение, подробнее об этом в главе «Создание и удаление файлов по инициативе файловой системы».



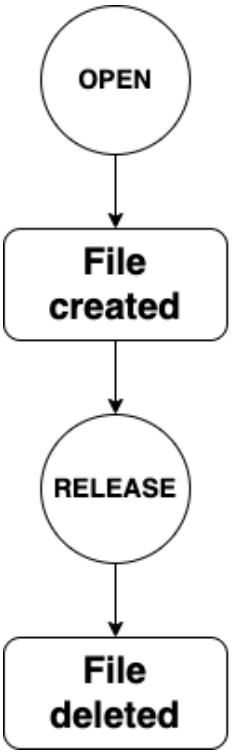
Жизненный цикл inode

Жизненный цикл dentry. Dentry — это жёсткая ссылка на inode. Первую такую ФС создаёт в момент генерации inode. Дополнительные жёсткие ссылки создаются при вызове link(). Жёсткая ссылка изменяется при вызове rename() и удаляется при вызове unlink(). Удаление жёсткой ссылки не приводит к немедленному удалению inode, потому что на inode без жёстких ссылок всё ещё могут ссылаться открытые файловые дескрипторы.



Жизненный цикл dentry

Жизненный цикл file. ФС создаёт файл при вызове open() или opendir(). Ссылка на него передаётся в каждую операцию над открытым файлом через поле `fi->fh`. При дублировании файлового дескриптора обе копии ссылаются на один и тот же `file`. Linux сам поддерживает счётчик ссылок на `file`, его не надо считать внутри ФС. Linux вызывает release() или releasedir(), когда счётчик ссылок опускается до нуля. В этот момент ФС удаляет файл.



Стандартные блокировки в файловой системе

Файловые системы предоставляют 2 стандартных механизма блокировок.

POSIX-блокировки. Устанавливаются с помощью fcntl(). Поддерживают блокировку частей файла. Поддерживают эксклюзивные (F_WRLCK, для операций записи) и разделяемые (F_RDLCK, для операций чтения) блокировки. Хранятся в контексте inode, это значит, что при установке блокировки с последующим созданием дочернего процесса, только родительский процесс будет владеть блокировкой.

BSD-блокировки. Устанавливаются с помощью flock(). Поддерживают только блокировки на файл целиком. Поддерживают эксклюзивные и разделяемые блокировки. Хранятся в контексте file, это значит, что при установке блокировки с последующим созданием дочернего процесса, и родительский, и дочерний процесс будут владеть блокировкой.

Оба вида блокировок - advisory. Это значит, что если процесс 1 установил блокировку, а процесс 2 решил изменить файл, ОС позволит процессу 2 внести изменения. Блокировки будут иметь силу, только если процесс 2 решит проверить наличие блокировки перед внесением изменений. В этом случае говорят, что процесс “уважает блокировки”.

Свои блокировки писать не обязательно. Если вы их не реализуете, то ОС будет использовать реализацию по умолчанию. Кастомные реализации блокировок имеют смысл для сетевых ФС, работающих с удалённым хранилищем.

Инструменты хранения состояния

Приступая к разработке ФС, надо ответить себе на несколько вопросов.

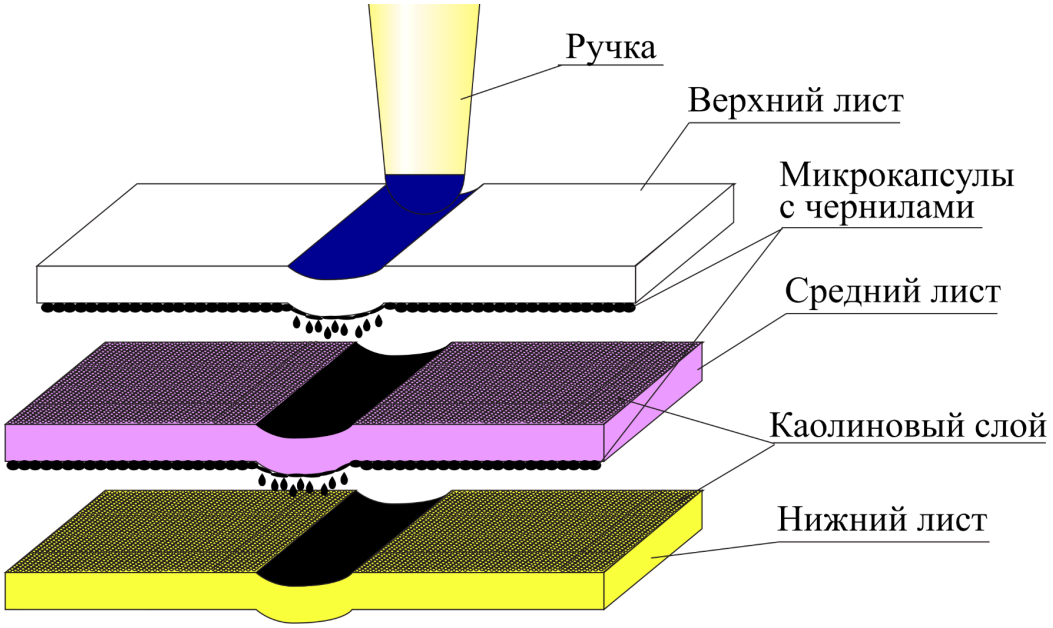
- Какую задачу вы хотите решить?
- Должна ли ваша ФС быть сетевой или локальной?
- Как хранить локальные данные?

Хранение локальных данных — важный аспект архитектуры вне зависимости от типа ФС. Все inode, dentry и file — это то состояние, которое ФС должна хранить и обслуживать. Сделать это можно разными инструментами.

Распространённый подход к хранению локальных данных — проксирование запросов к нижележащей файловой системе. В этом варианте FUSE-файловая система сохраняет данные на диск через ФС общего назначения. Такие прокси-ФС называются «сквозными», passthrough.

passthrough-файловые системы делают opendir на директорию, в которую будут монтироваться. Так они получают доступ к нижележащей ФС. passthrough-файловые системы

используют нижележащую ФС как хранилище для своих локальных данных. Такой подход упрощает разработку, так как не требует взаимодействия с устройствами жёстких дисков напрямую. При этом снижается производительность. Если вам интересны passthrough-ФС, обратите внимание на оптимизации в этом направлении, подробнее о них в разделе «Производительность».



Passthrough файловые системы похожи на копировальную бумагу: верхний лист - это passthrough ФС, все записи в итоге сохраняются на нижний лист - нижележащую ФС

Если passthrough для вас не подходит, придётся реализовывать свои структуры для хранения состояния: деревья, хеш-таблицы и т. п. Проще всего взять встраиваемую СУБД и сохранять все inode, dentry и file в неё, там необходимые алгоритмы уже реализованы.

В своём проекте я использовал Tarantool с отключённой записью на диск для хранения inode, dentry и file. Для содержимого файлов я использовал memfd — его интерфейс совместим с FUSE-запросами read() и write(). Таким образом, я держу все состояние в оперативной памяти — это нестандартное решение. Оно позволяет добиться высокой производительности, но не подходит для долговременного хранения.

Заключение

Зачем я дал столько теории? Вся эта информация находится в разрозненных источниках, а я был бы очень рад прочитать подобную статью, когда начинал работу над проектом. Если вы хотите разработать свою файловую систему, следующим шагом скопируйте к себе и соберите пример из репозитория libfuse. Дополняйте код в соответствии со своей задачей, и возвращайтесь к тексту выше как к справочнику, когда у вас появятся вопросы.

II. Подводные камни разработки и эксплуатации FUSE-файловой системы

Здесь мы переходим от теории к практике, рассмотрим проблемы, с которыми столкнулись при эксплуатации FUSE в продакшене, и как их решали. Если читаете с самого начала, спасибо, что добрались сюда. Возможно, стоит сделать паузу и передохнуть 🙂

Инструменты администрирования FUSE

Давайте рассмотрим, какие инструменты есть для эксплуатации FUSE, и какие проблемы решаются с их помощью.

Настройка /etc/fuse.conf

`/etc/fuse.conf` — это конфиг для контроля доступа к FUSE. Доступ к FUSE-файловым системам ограничен, чтобы пользователи не вносили в дерево файлов неожиданных изменений, пользуясь непривилегированным монтированием. Это ограничение не связано с правами доступа к файлам. Оно реализовано на уровне FUSE-модуля ядра и защищает от атак вида “примонтировать свою ФС в /bin, подменить исполняемый файл и эскалировать привилегии”. По умолчанию, к FUSE-файловой системе может получить доступ только её владелец — пользователь (UID/GID), который примонтировал файловую систему. Системный администратор может ослабить ограничения и открыть доступ для других пользователей, для этого в `/etc/fuse.conf` есть опции:

- `user_allow_root` — позволяет root-пользователю получать доступ к FUSE-файловым системам, владельцем которых он не является;
- `user_allow_other` — позволяет всем пользователям получать доступ к FUSE-файловым системам, владельцами которых они не являются.

FUSE ограничивает количество одновременно примонтированных файловых систем. За это отвечает опция `mount_max = NNN`, по умолчанию 1000.

В `/etc/fuse.conf` можно указывать стандартные опции монтирования для всех FUSE-файловых систем. Подробнее — в руководстве.

FUSE control filesystem

FUSE control filesystem — это инструмент для управления FUSE-файловыми системами. Выполните `mount -t fusectl none /sys/fs/fuse/connections`, чтобы примонтировать файловую систему. В директории `/sys/fs/fuse/connections` вы увидите список активных FUSE-соединений.

```
Unset
ls -1A /sys/fs/fuse/connections
321
```

Каждое соединение — директория с несколькими файлами.

```
Unset
ls -1A /sys/fs/fuse/connections/321
abort
congestion_threshold
max_background
waiting
```

Как сопоставить экземпляр ФС и ID FUSE-соединения в FUSE control filesystem? Ответ спрятан глубоко, но он есть: нужно смотреть на номер устройства примонтированной файловой системы:

```
Unset
cat /proc/self/mountinfo | grep "\- fuse /dev/fuse" | awk '{print $3 " " $5}'
0:321 /var/spool/exim/input
#^ это одно число, в формате MAJOR:MINOR, может понадобиться конвертация
```

Сравните три листинга кода выше. FUSE-файловая система с ID соединения `321` примонтирована по пути `/var/spool/exim/input`.

Запись любых данных в файл `/sys/fs/fuse/connections/321/abort` разрывает FUSE-соединение. Это самый жёсткий способ отмонтировать файловую систему, он работает, даже когда `kill -9` не проходит.

Три других файла нужны для работы с очередями запросов внутри FUSE-модуля ядра. Их назначение:

- `waiting` — чтение из файла показывает количество запросов в очередях FUSE. Файл не доступен для записи.
- `max_background` — ограничивает количество параллельных асинхронных запросов (FORGET), чтобы ФС хватало производительности обрабатывать синхронные. Файл доступен для чтения и записи.
- `congestion_threshold` — пороговое значение длины очередей, при котором vfs начинает троттлить запросы к файловой системе. Файл доступен для чтения и записи.

Подробнее об очередях FUSE написано в этой статье.

На практике я пользовался только файлом `abort`, про пользу от тонкой настройки очередей FUSE ничего сказать не могу.

Автоматическое монтирование

Если вы используете FUSE в продакшене, вероятно, вы захотите монтировать ФС автоматически при запуске ОС. Обратите внимание на fstab и systemd.automount. У меня FUSE эксплуатируется в Kubernetes, поэтому необходимости разбираться с автоматическим монтированием не было.

Монтирование FUSE в Kubernetes

Чтобы примонтировать FUSE в K8s-контейнере, вам нужно:

- разрешение на `mount` в seccomp-профиле — есть по умолчанию;
- доступ к устройству `/dev/fuse` — можно дать с помощью volume hostpath или k8s device plugin'а;
- capability CAP_SYS_ADMIN в текущем пространстве имён.

Чтобы выдать capability можно запустить контейнер с привилегиями, но это опасно. Вместо этого лучше создать новые пространства имён mount и user. В этом случае Linux даст вам примонтировать ФС без привилегий, но никто кроме вашего процесса и его дочерних процессов не увидит вашу ФС.

Пространства имён Linux — сложная тема, напишите в комментарии, если вам интересно прочитать про неё отдельную статью. Здесь я дам только решение, как безопасно примонтировать FUSE в K8s, без подробных объяснений, как оно работает. Воспользуйтесь одним из двух вариантов.

- С версии 1.27 Kubernetes умеет создавать user namespace для пода. Монтирование FUSE — один из способов использования этой фичи, подробнее в KEP-127.
- Если у вас более старая версия Kubernetes, создайте пространства вручную с помощью утилиты unshare. Запускайте контейнер с драйвером ФС так:

```
Unset
unshare -U -m -r -- /path/to/filesystem/binary --maybe-with-args
```

Создание отдельных пространств user и mount для вашей ФС позволяет примонтировать её без CAP_SYS_ADMIN.

Производительность

Вопрос о производительности FUSE стоял особенно остро, когда я только запускал систему. Звучали мнения, что запросы к FUSE-файловой системе могут занимать секунды — очень долго. Количество системных вызовов вырастает до x2 при работе с FUSE, это следствие архитектуры системы. Однако, само по себе переключение контекста при системном вызове

занимает не так уж много времени: 1-2 микросекунды. Более того, если вам мешает количество переключений контекста, есть целый ряд доступных оптимизаций:

- writeback cache;
- zero-copy с помощью сплайсинга;
- FUSE_PASSTHROUGH.

Исследование производительности FUSE подтверждает, что во многих вариантах нагрузки снижение производительности на FUSE не превышает 5 %. В исследовании есть случаи, в которых производительность падает существенно — до 83 %, но таких примеров немного.

FUSE сам по себе не будет тормозить вашу систему «на секунды», даже с «плохим» паттерном нагрузки. По моему опыту, производительность проседает чаще из-за кода обработчиков FUSE-запросов, чем из-за транспорта от клиента до ФС через ядро. Например, если ваши обработчики постоянно ждут блокировки или синхронно ходят в сеть — да, вы можете повиснуть на секунды. Если у вас проблемы с производительностью FUSE-ФС, убедитесь, что драйвер ФС работает эффективно.

Надеюсь, у меня получилось развеять предубеждения относительно производительности FUSE. В следующей главе рассмотрим модель многопоточности: как спроектировать архитектуру ФС, чтобы минимизировать проблемы с производительностью.

Модель многопоточности

Три самых частых ошибки при разработке своей ФС:

1. Deadlock.
2. Use after free.
3. Просадка производительности.

Количество ошибок во всех трёх случаях можно уменьшить на этапе выбора модели многопоточности. В этой главе я расскажу о том, к какой модели я пришёл в своём проекте.

Для хранения inode и dentry я использую Tarantool. Это СУБД и сервер приложений на C/Lua. Моя ФС — это Tarantool-приложение, поэтому модель многопоточности ФС в моём случае тесно связана с моделью многопоточности Tarantool.

В Tarantool к данным имеет доступ строго один поток. В нём выполняются файберы — асинхронные созадачи (корутины). Файберы не блокируют поток выполнения, любая блокирующая операция передаёт управление готовому к выполнению файберу. Алгоритм диспетчеризации файберов невытесняющий. Это значит, что если файбер получил управление, он может выполняться бесконечно, пока сам не решит вернуть управление. Операции ввода-вывода вынесены в отдельный пул потоков. Благодаря такой модели многопоточности Tarantool гарантирует линеаризованный доступ к данным — псевдопараллельные файберы выполняют операции над данными по-честному

последовательно. Это хорошая база для файловой системы: ситуации гонки исключены по умолчанию, не надо обкладывать структуры данных мьютексами.

Мы интегрировали цикл обработки событий libfuse с файберами Tarantool. У libfuse для этого есть API самописных циклов обработки событий, подробнее писал про него в главе «Сессия libfuse». При монтировании файловой системы мы запускаем фоновый файбер, ответственный за обработку запросов FUSE. Он работает по алгоритму:

1. переводит FUSE-соединение в неблокирующий режим;
2. ждёт, когда соединение будет доступно на чтение;
3. читает FUSE-запрос из соединения в буфер с помощью fuse_session_receive_buf() ;
4. парсит FUSE-запрос в буфере и выполняет соответствующий обработчик с помощью fuse_session_process_buf().

Получается такая модель:

- FUSE-запросы обрабатываются в один поток, в том порядке, в котором они пришли;
- чтение из FUSE-соединения — неблокирующее;
- обработчики — синхронные и тоже неблокирующие, благодаря линеаризованному доступу к данным Tarantool.

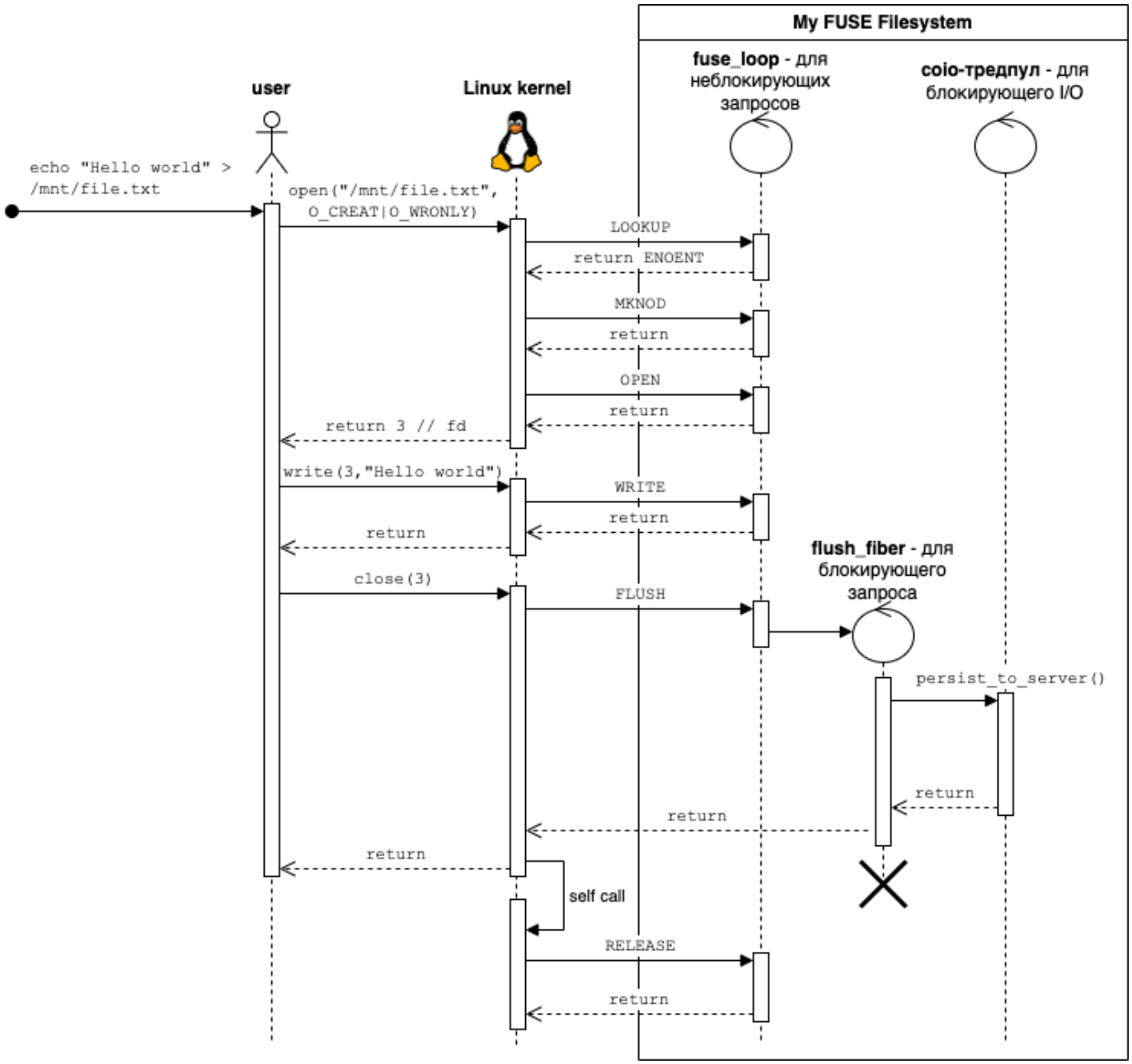
Эта модель позволяет эффективно обрабатывать FUSE-запросы, без затрат времени на блокировки в общем случае.

Асинхронные обработчики. Любой блокирующий обработчик плохо укладывается в описанную модель: он приводит к проблеме Head of line blocking, блокирует цикл обработки событий libfuse. В блокирующих обработчиках мы запускаем отдельный файбер для обработки запроса.

В каких случаях обработчики получаются блокирующими?

- SETLK — установка на файл Posix-блокировки, блокирующей по определению. Подробнее в главе «Стандартные блокировки в файловой системе».
- FLUSH — моя файловая система — сетевая, в этом обработчике она отправляет файл на сервер по сети. Подробнее в главе «Как детектировать окончание работы с файлом».
- FORGET — в этом обработчике моя ФС удаляет файл с сервера по сети. Подробнее в главе «Жизненный цикл структур данных файловой системы».

Неблокирующая синхронная обработка FUSE-запросов в один файбер и отдельные файберы для блокирующих обработчиков. К этой модели я в итоге пришёл, пока что полёт нормальный — можно пользоваться.



Модель многопоточности в действии, на примере простой операции над файловой системой

Два вида взаимной блокировки

Предположим, у нас есть два процесса, потока или корутины: А и Б. Они хотят заблокировать два ресурса: Р1 и Р2. А заблокировал Р1, Б — Р2. Теперь ни А, ни Б не смогут заблокировать оба ресурса и будут бесконечно долго ждать друг друга — произошла взаимная блокировка (deadlock).

Deadlock — хорошо известная тема. Написано достаточно теоретических материалов про их предотвращение и обработку. В этом разделе обсудим практический опыт: как обнаружить взаимные блокировки, чем они отличаются и как восстановить систему.

Deadlock в user space

Симптомы. Некоторые процессы зависают при запросе к вашей ФС. В списке процессов видны зависшие, по их логам можно оценить, что дело в ФС.

Как отлаживать. Читайте логи ФС. В моей не так много блокировок, поэтому я добавил лог на взятие каждой из них. По таким логам видно, какой актор какой ресурс ждёт.

Как лечить. Перезапустите вашу ФС, `kill` сработает. Я в таких случаях переписывал код, чтобы исключить возможность взаимной блокировки.

Deadlock в kernel space

Симптомы. Все процессы зависают при запросе к вашей ФС. В списке процессов видно, что драйвер ФС висит в состоянии uninterruptible sleep. В этом случае у вас не получится ни `kill -9`, ни `gdb -p`.

Как отлаживать. У меня получилось снять бэктрейс с помощью `cat /proc/<pid>/stack`. По логам ФС можно найти, что все проблемы с такими симптомами имеют одинаковый характер: системный вызов не вернул управление.

Как лечить. Перезапустите ФС, разорвав FUSE-соединение через FUSE control filesystem.

Все мои блокировки в kernel space были связаны с использованием fuse_lowlevel_notify_* API. Я вызываю эти функции строго из асинхронных обработчиков, подробнее в главах «Модель многопоточности» и «Создание и удаление файлов по инициативе файловой системы». Взаимные блокировки ушли, когда я начал выполнять функции `fuse_lowlevel_notify_*` в отдельном thread pool с помощью coio_call.

Создание и удаление файлов по инициативе файловой системы

Сетевые файловые системы получают изменения в файлах с сервера. Это значит, что файлы в такой ФС могут создаваться и удаляться «по инициативе ФС», то есть без действий клиентов на локальной машине. В этом разделе рассмотрим, как ФС может сказать ядру, что файл был создан или удалён.

Создание файлов

Чтобы создать файл, достаточно сохранить inode и dentry в локальное хранилище ФС. Клиент увидит ваш файл, когда запросит список файлов в директории.

Существует разница между inode, созданной с помощью mknod, и inode, которую ФС просто вставила в локальное хранилище. Напомню что у inode есть два счётчика ссылок (подробнее об этом в главе «Жизненный цикл структур данных файловой системы»):

- `nlookup` — сколько структур данных в ядре Linux ссылается на inode;

- `nlink` — сколько жёстких ссылок в файловой системе ссылается на inode.

При создании inode и dentry с помощью mknod:

- `nlookup = 1` — потому что ответ fuse_reply_entry() на запрос mknod() требует инкрементировать `nlookup`;
- `nlink = 1` — потому что запрос mknod() создаёт новую dentry, жёсткую ссылку на inode.

При вставке inode и dentry в локальное хранилище по инициативе ФС, без запроса от ядра:

- `nlookup = 0` — потому что никакие структуры данных в ядре ОС не ссылаются на inode, ядро о ней ничего не узнает, пока не перечитает директорию;
- `nlink = 1` — потому что ФС создала новую dentry.

Я не уверен, что такое поведение правильное, но оно работает для моей ФС. Вариант `nlink > nlookup` стоит учитывать в обработчике forget(). Обычно удаление inode происходит асинхронно, когда падает `nlookup` до нуля. Если ваша ФС может создавать ситуации `nlink > nlookup`, то `nlink` тоже нужно проверять перед удалением inode.

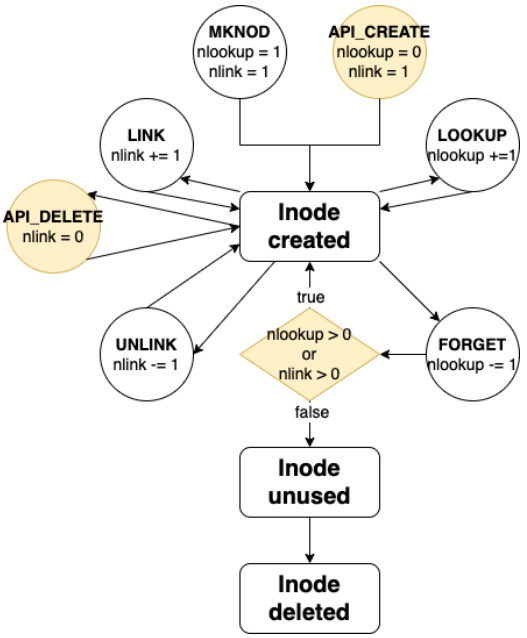
Удаление файлов

Удалить файлы по инициативе ФС труднее, чем создать. В ядре ОС могут оставаться ссылки на ваши структуры данных, поэтому ядро надо уведомить, что они удалены. Алгоритм удаления файла по инициативе ФС:

1. выбираем inode, которую хотим удалить;
2. находим все жёсткие ссылки (dentry) на эту inode;
3. удаляем каждую найденную dentry из локального хранилища;
4. для каждой удалённой dentry вызываем fuse_lowlevel_notify_delete().

Не удаляйте inode после шага 4, она всё ещё может использоваться открытыми файлами. Inode удалится асинхронно, когда ОС пришлёт все FORGET'ы.

Вызывайте все функции `fuse_lowlevel_notify_*` в отдельном потоке, который не занимается обработкой FUSE-запросов, чтобы не спровоцировать deadlock в kernel space.



Жизненный цикл inode с учётом API_* операций по инициативе файловой системы, жёлтым отмечена разница с жизненным циклом в общем случае

Как обнаруживать окончание работы с файлом

Сетевые файловые системы синхронизируют содержимое файлов с сервером. Моя ФС отправляет все изменения в файле за один запрос к серверу, когда пользователь закончил работу с файлом. Такой подход позволяет объединять результаты нескольких операций write() в один сетевой запрос.

Один из вариантов обнаружить окончание работы с файлом — отслеживать, когда он закрывается. Закрытие файла в FUSE разделено на два запроса: flush() и release(). Рассмотрим их подробнее.

FLUSH. Вызывается каждый раз, когда закрывается файловый дескриптор. Файловые дескрипторы могут дублироваться, поэтому на один вызов open() может приходиться несколько вызовов `flush()`, например, когда процесс открывает файл и затем создаёт дочерние процессы. При вызове close() операционная система вызывает `flush()` синхронно, это позволяет вернуть ошибку, если сохранить файл на сервер не удалось. И да, этот запрос никак не связан с fflush() из glibc 🙂

RELEASE. Вызывается один раз, когда закрывается последний файловый дескриптор. Этот вызов нужен, чтобы удалить файловое описание, a.k.a `file`. Подробнее об этом в главе «Структуры данных файловой системы». На один `open()` всегда будет только один `release()`. Хотя API `release()` позволяет вернуть ошибку, она не будет возвращена при вызове `close()`. Более того, попытки вернуть ошибку из обработчика `release()` в моём случае приводили к ошибкам при взаимодействии с ядром ОС. Лучше использовать этот запрос только если вы используете stateful I/O, как деструктор для `file`.

Я использовал обработчик `flush()` для обнаружения окончания работы с файлом. Про него в документации сказано страшное:

> ** Filesystems shouldn't assume that flush will always be called*
> ** after some writes, or that if will be called at all.*

Подробнее этот аспект разъяснён в этом e-mail. FLUSH даёт такие гарантии:

- FLUSH гарантированно будет вызван на каждый `close()`;
- FLUSH не обязательно означает, что файл был изменён;
- FLUSH может быть вызван несколько раз, если файловый дескриптор был дублирован;
- FLUSH будет вызван ДО окончания работы с файлом в случае: open() → mmap() → close() → `/* modify the file mapping */` → munmap().

В моём случае клиенты гарантированно не пользуются `mmap()`, поэтому я считаю FLUSH окончанием работы с файлом и сохраняю файлы на сервер в обработчике `flush()`. Дублирующие вызовы `flush()` всё ещё возможны, но обновить файл на сервере лишний раз - не страшно. Можно помечать inode флагом при внесении изменений, и обновлять файл на сервере только если содержимое было изменено.

Тестирование

Если вы пишете файловую систему, вероятно, вы хотите быть совместимы с ФС общего назначения, насколько это возможно. Для проверки совместимости я использовал набор тестов pjdfstest. Коллеги из Яндекса использовали помимо него ещё несколько тестов, в том числе нагрузочные.

Мониторинг

Логи доступа. Это первый инструмент мониторинга, который вам понадобится. Libfuse пишет неплохие логи, их можно включить опцией `-odebug` при создании сессии libfuse. Самописный логгер можно использовать в новых версиях libfuse, см. fuse_set_log_func(). В старой версии я руками прописал логи доступа в своих обработчиках.

Что должно быть в логах:

- запрос с ключевыми аргументами;
- ответ, код ошибки при наличии;
- сквозной ID запроса — сложите случайную строку в переменную `thread-local` и используйте в каждом логе.

Метрики. Следим за временем выполнения и пропускной способностью FUSE-обработчиков. Строим графики с разбивкой по типу запроса, errno-коду ошибки. При нормальной работе ФС обработчики не должны возвращать ошибки, но есть исключения:

- `lookup()` может возвращать ENOENT, «файл не найден» — это корректный ответ;

- `setlk()` может возвращать EAGAIN, «файл заблокирован другим процессом» — тоже корректно.

Во всех случаях, кроме обозначенных исключений, я держу нулевой порог оповещений на количество ошибок.

III. Заключение

В статье я изложил три года опыта разработки файловой системы. Надеюсь, мне удалось создать самый глубокий материал про FUSE в рунете, и вам он будет полезен. Не бойтесь разрабатывать файловые системы — это увлекательное занятие.

24 и 25 июня пройдёт конференция Saint Highload++, на ней я расскажу подробнее, какую задачу мы решали с помощью FUSE и какую роль наша файловая система играет в проекте Mail.ru. Приходите послушать доклад и пообщаться! Подписывайтесь на мой Telegram-канал, там я пишу о теории и практике разработки ПО.

Остались вопросы? Буду рад ответить в комментариях)