

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

Реализация RAID5 в фреймворке SPDK

Байтенов Арсений Леонидович

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
инженер-исследователь лаборатории технологий программирования инфраструктурных
решений СПбГУ А. И. Васенина

Санкт-Петербург
2024

Оглавление

Введение	4
1. Постановка задачи	6
1.1. Осенний семестр	6
1.2. Весенний семестр	6
2. Обзор	7
2.1. RAID	7
2.1.1. RAID0	7
2.1.2. RAID5	8
2.2. SPDK	9
2.2.1. Минимизация операций копирования данных . . .	9
2.2.2. Передача сообщений	10
2.3. Аналогии	11
2.3.1. RAID0 в SPDK	11
2.3.2. mdadm: RAID5	11
2.3.3. OpenZFS: ZRAID	11
3. Реализация	13
3.1. Структуры запроса	13
3.2. Система функций и callback-функций для реги- запросов чтения и записи в RAID5	14
3.3. Обработка запросов чтения	15
3.3.1. Без битых стрипов	16
3.3.2. Один битый стрип	16
3.4. Обработка запросов записи	16
3.4.1. Read-modify-write	17
3.4.2. Битый стрип паритета	18
3.4.3. Битый стрип запроса	18
3.5. Выделение памяти буфферов запроса с глобальной адре- сацией	18
3.5.1. RAID5 io channel	19

3.6. Обработка запросов в многопоточном режиме	20
4. Тестирование	21
Заключение	22
Список литературы	23

Введение

На уровне планирования процессорного времени, обработка запросов чтения/записи происходит асинхронно. Поток посылает запрос блочному устройству, происходит смена контекста, и вычислительные ресурсы передаются другому потоку. Когда блочное устройство обрабатывает запрос, или во время обработки возникнет ошибка, вызовется прерывание, опять сменится контекст, и ресурсы будут переданы потоку, инициировавшему запрос. Между вызовом запроса и моментом, когда придёт ответ, проходит довольно много времени, то есть оптимизация работы с блочными устройствами может существенно улучшить скорость работы программы.

Операцию, которая обращается к длинной непрерывной области памяти, можно было бы обрабатывать параллельно, если бы эта область памяти на самом деле располагалась на нескольких блочных устройствах. Отсюда рождается идея RAID-массивов¹, массивов, объединяющих базовые блочные устройства в единое пространство хранения данных. Иными словами, с целью повышения производительности и отказоустойчивости диски объединяются в один логический модуль, который для внешней среды представляется как новое блочное устройство. RAID-массивы классифицируются по разным уровням, которые предоставляют различные преимущества за некоторую потерю в полезном объёме памяти. RAID0 представляет из себя простое объединение блочных устройств, не гарантирующие какую-либо отказоустойчивость, тем не менее позволяющее ускорить обработку запросов чтения/записи, за счёт того, что запрос к непрерывной снаружи области памяти, на самом деле выполняется сразу несколькими блочными устройствами одновременно. RAID1 дублирует данные на всех дисках, тем самым гарантируя возможность восстановления информации, если есть хотя бы одна целая копия. RAID5 представляет из себя RAID0 с контрольно-восстановительными суммами, кроме ускорения обработки запросов предоставляет возможность восстановить данные в случае вы-

¹Redundant Array of Independent Devices.

хода из строя одного базового блочного устройства. Пожертвовать приходится пространством памяти суммарно равным размеру одного базового блочного устройства. Именно эта память тратится на контрольно-восстановительные суммы, хранящиеся на разных базовых блочных устройствах. Реализации RAID-массивов бывают как аппаратные, так и программные. Аппаратная реализация имеет более высокую стоимость, однако контроллер выполняет вычисления самостоятельно, снижая нагрузку на процессор. Программный аналог может требовать конкретную операционную систему, но предоставляет возможность обновления и настройки без приобретения дополнительного оборудования, а также более широкие возможности оптимизации.

С развитием технологий скорость обработки запроса диском возросла, таким образом, задержка на переключение контекста процессора стала чувствительней. Аппаратное обеспечение больше не является узким местом, наоборот оно сместилось в сферу программных механизмов, так как, кроме прочего, из-за взаимодействия с блочными устройствами через пространство ядра приходится выполнять операции копирования данных запросов из пространства ядра в пространство пользователя. Решением этого вопроса в фреймворке SPDK² стала возможность работать с блочными устройствами через пространство пользователя. По performance reports компании Intel [3], это делает работу СХД быстрее.

В SPDK есть реализации RAID0 и RAID1, однако нет реализации RAID5. Таким образом, целью работы является добавление функциональности RAID5 в фреймворк SPDK и сравнение с аналогами.

²[Storage Performance Development Kit](#).

1 Постановка задачи

Целью работы является реализация RAID5 в фреймворке SPDK. Для её выполнения были поставлены следующие задачи:

1.1 Осенний семестр

1. Спроектировать систему функций и callback-функций для регистрации запросов чтения и записи в RAID5.
2. Реализовать RAID5.
3. Написать интеграционные тесты для проверки корректности полученной реализации.

1.2 Весенний семестр

1. Реализовать выделение памяти буфферов запроса с глобальной адресацией RAID5.
2. Реализовать обработку запросов в многопоточном режиме.
3. Проверить корректность полученной реализации.

2 Обзор

2.1 RAID

RAID — Redundant Array of Independent Devices — технология хранения данных, основанная на объединении базовых устройств хранения в массив, единый логический модуль, представляющий из себя новое блочное устройство. Смысл такого объединения заключается в повышении скорости обработки запросов и(или) отказоустойчивости. Для понимания последующего рассказа будут введены некоторые термины. Для сопоставления адресов виртуального устройства с адресами базовых устройствам используются стрипы [4], области памяти на базовых устройствах. Стрипы адресуются последовательно на каждом базовом устройстве и имеют одинаковый объём памяти. Страйп [4] представляет из себя набор стрипов в соответствующих местах базовых устройств. Неформально его можно описать как один «уровень» стрипов в массиве базовых устройств. Отношение между понятиями стрип и страйп графически представлено на рис. 1. Для восстановления потерянной информации некоторые уровни RAID-массивов предполагают хранение контрольно-восстановительных сумм на базовых устройствах. Стрип, в котором эти суммы хранятся называется стрипом паритета. Расположение таких стрипов может быть разным, пример расположения стрипов паритетов RAID5 приведён на рис. 2.

2.1.1 RAID0

RAID0 — RAID-массив, в котором последовательные логические блоки данных равномерно распределяются по набору независимых базовых устройств. Такое распределение данных позволяет увеличить скорость обработки запросов чтения и записи за счёт распараллеливания запроса, так как подряд идущие данные находятся на разных устройствах. RAID0 не содержит какой-либо информации для восстановления потерянных данных, что может привести к потере данных при выходе из строя хотя бы одного базового устройства. Изображение RAID0 приве-

дено на рис. 1.

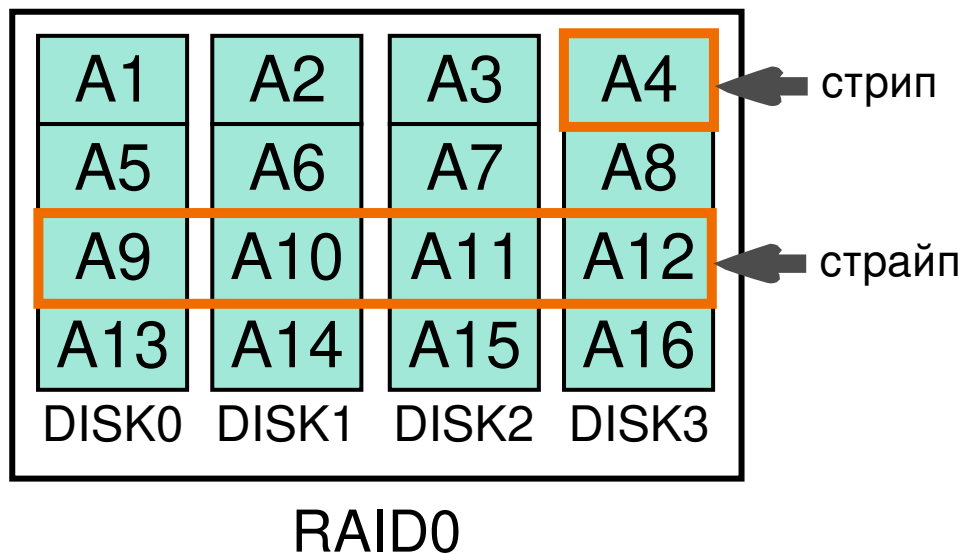


Рис. 1: RAID0 на четырёх базовых устройствах.

2.1.2 RAID5

RAID5 отличается от RAID0 тем, что в каждом страйпе один из стрипов отводится под данные паритета. В этом стрипе хранится результат операции XOR стрипов с данными. Таким образом, теряется объём данных, суммарно равный объёму одного базового устройства. RAID5 проигрывает RAID0 не только по объёму полезных данных, но и по скорости обработки запроса из-за необходимости обновления стрипов паритета. За счёт этих модификаций RAID5 предоставляет возможность работы при выходе из строя не более чем одного стрипа в страйпе. Изображение RAID5 приведено на рис. 2.

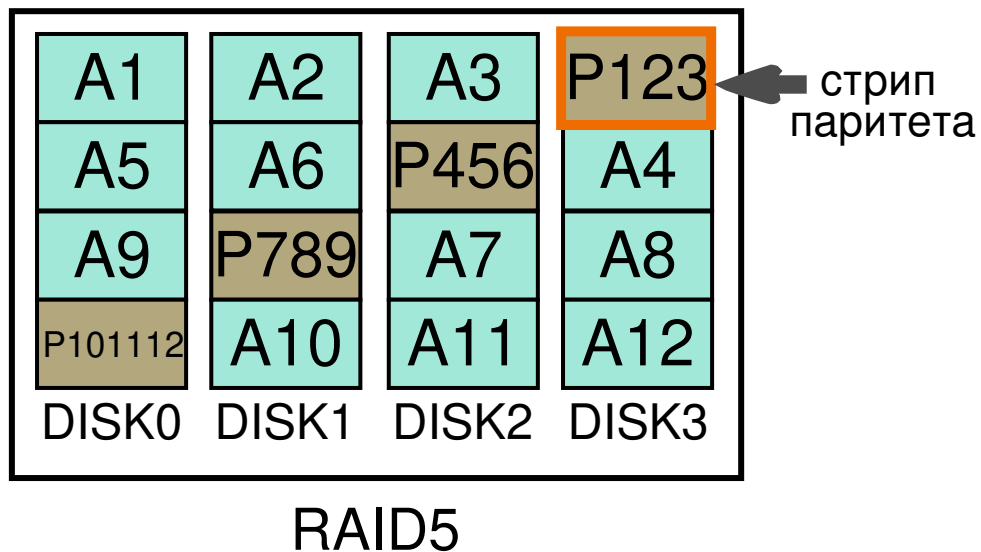


Рис. 2: RAID5 на четырёх базовых устройствах.

2.2 SPDK

SPDK — Storage Performance Development Kit — фреймворк с открытым исходным кодом для разработки высокопроизводительных пользовательских приложений хранения данных. Набор инструментов и библиотек разработан компанией Intel.

2.2.1 Минимизация операций копирования данных

Особенностью SPDK является взаимодействие с блочными устройствами через пространство пользователя. Одним из преимуществ фреймворка является возможность использования буфферов, выделяемых пользователем для отправки запросов на устройства, поддерживающие DMA, технологию прямого доступа к памяти без участия центрального процессора. То есть драйвер устройства, поддерживающего DMA, например SSD NVMe, работающий в пространстве пользователя, минимизирует количество операций копирования из kernel space в user space [5]. Кроме того, драйвер NVMe в SPDK, опрашивает устройство во время

запроса чтения/записи. Современные SSD NVMe успевают обработать запрос до этого момента, то есть не приходится переключать ресурсы процессора на другие потоки или процессы [7].

2.2.2 Передача сообщений

Традиционно, для достижения параллелизма общие ресурсы защищаются примитивами синхронизации. Таким образом, при использовании общих ресурсов поток блокирует их, заставляя тем самым остальные потоки дожидаться разблокировки. Это позволяет легко преобразовать однопоточные приложения в многопоточные, обернув общие ресурсы с использованием примитивов синхронизации. Также с точки зрения программиста программы исполняются как последовательность императивных инструкций. Тем не менее, с увеличением числа потоков, увеличивается число обращений к общим данным, а следовательно и число блокировок, заставляя потоки простаивать.

Одной из основных целей SPDK является линейное масштабирование с добавлением аппаратного обеспечения. Для достижения этой цели должны быть независимы друг от друга настолько, насколько это возможно. Это означает минимизацию числа блокировок общих ресурсов и использования атомарных операций.

SPDK не использует традиционный подход с блокировкой ресурсов, вместо этого заводится один поток, взаимодействующий с ресурсами. Когда другим потоком нужно провзаимодействовать с данными, они отправляют сообщения владеющими этими данными потоку, а тот в свою очередь выполняет операции от их имени. В сообщении указывается функция обратного вызова и контекст, для продолжения обработки после взаимодействия с данными. Эта схема позволяет эффективней использовать кэш, располагая данные в кэше ближе к ядру владеющего потока, не говоря уже об снижении расходов на синхронизацию кэша для разных ядер по сравнению с традиционной моделью.

2.3 Аналоги

В SPDK реализован только нулевой уровень RAID-массивов, а именно RAID0. В то время как, непосредственными открытыми программными аналогами RAID5 являются решения, предоставляемые утилитой mdadm внутри ядра Linux и файловой системой OpenZFS.

2.3.1 RAID0 в SPDK

RAID0 в SPDK содержит обработку запросов чтения/записи к одному стрипу. Она реализована через делегирование запроса соответствующему базовому блочному устройству, посредством пересчёта логического адреса стрипа в физический. RAID0 не содержит каких-либо вычислений паритетных данных или выделений памяти, передавая данные запроса такими, какими получает. Таким образом, эта реализация не ограничивает поддержку DMA.

2.3.2 mdadm: RAID5

Утилита mdadm позволяет создать RAID5, однако драйвер такого RAID-массива работает в пространстве ядра, то есть обработка запросов происходит в kernel space. Это ведёт к необходимости копировать данные запросов из пространства ядра в пространство пользователя, а также к задержкам на переключение контекста процессора. Кроме того, недостатком этой реализации являются глобальные блокировки [6], например при пересчёте паритетов, что опять же снижает производительность.

2.3.3 OpenZFS: ZRAID

OpenZFS — платформа хранения данных, включающая в себя функциональность файловой системы. В OpenZFS нет реализации RAID5, однако есть решение сравнимое с ним — ZRAID [2]. Первое отличие — это использование технологии “copy-on-write”, идея которой заключается в хранении копий запросов записи в незанятой памяти. В некоторый

момент времени система принимает попытку атомарной записи части данных в соответствующее место. Такая технология защищает данные от внезапного сбоя всего ZRAID. Вторым отличием от RAID5 является нефиксированный размер страйпа, ведь благодаря файловой системе появляется знание, какая память занята, в отличие от RAID5, вынужденного гарантировать сохранность всех данных на дисках. Тем не менее, драйвер ZRAID всё равно работает в kernel space, что ведёт к необходимости копировать данные запросов из пространства ядра в пространство пользователя.

3 Реализация

Реализация была написана на языке C, поскольку в фреймворке SPDK интерфейс для встраивания новых уровней RAID-массивов реализован на языке C. Чтобы встроить новый модуль, необходимо инициализировать структуру, ассоциированную с данным модулем. Кроме уровня RAID-массива и минимального количества базовых устройств, необходимо указать, по крайней мере, две функции: функцию старта устройства(`raid5_start`) и функцию регистрации запроса чтения/записи(`raid5_submit_rw_request`). В функции старта текущей реализации взводится флаг, указывающий на требование разбивать запросы для модуля на запросы размером не больше страйпа. Функция регистрация запроса чтения/записи принимает на вход указатель на структуру `raid_bdev_io`, ассоциированную с запросом к RAID-массиву. Однако эта структура не предполагает наличие каких-либо временных буферов для хранения данных, которые будут читаться из или записываться в несколько отдельных базовых устройств.

3.1 Структуры запроса

Для хранения буферов ассоциированных со страйпом, используется структура `chunk`, в которой кроме массива буферов хранится флаг `enabled`, обозначающий включён ли этот страйп в запрос.

С запросом к страйпу ассоциирована структура `stripe_request` кроме прочего хранящая массив структур `chunk`, представляющий страйп как набор стрипов, и массив буферов выделенных и используемых в качестве буферов стрипов не используемых в запросе. Связи структур, нужных для обработки запроса можно посмотреть на рис. 3.

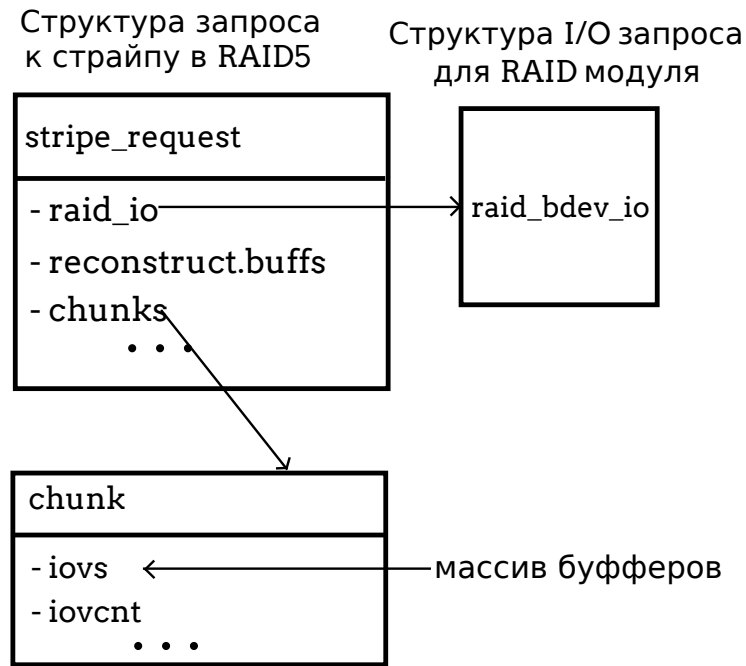


Рис. 3: Структура запроса чтения/записи к страйпу.

3.2 Система функций и callback-функций для регистрации запросов чтения и записи в RAID5

В силу особенности SPDK передачи сообщений для каждого обращения к данным RAID-массива необходимо описать не только функцию для отправки запроса чтения/записи к базовым блочным устройствам, но и функцию обратного вызова, для последующего обработки сценарии, которая будет вызвана после взаимодействия с данными.

Для минимизации объёма исходного кода непосредственное делегирование запросов в базовым блочным устройствам вынесено как отдельная логика. В ней участвуют:

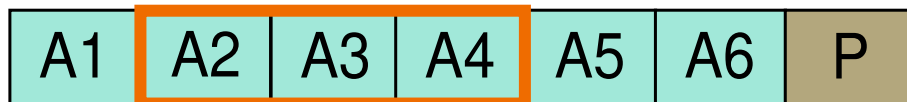
- `raid5_submit_stripe_request` — для установки типа запроса и функции обратного вызова;
- `raid5_submit_chunk` — для непосредственной отправки запроса базовому устройству;

- `raid5_submit_chunks` — для вызова предыдущей функции к нужным стрипам;
- `raid5_submit_chunk_retry` — callback-функция для повторной отправки запроса в случае нехватки памяти;
- `raid5_chunk_complete_bdev_io` и `raid5_chunk_complete` — callback-функции для вызова функций дальнейшей обработки в случае успешного завершения всех запросов.

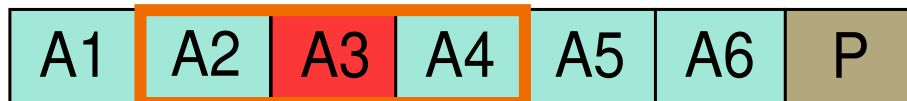
3.3 Обработка запросов чтения

Для обработки запросов чтения из функции `raid5_submit_rw_request` вызывается функция `raid5_submit_read_request`, которая нужна, чтобы определить сценарий и выделить память для буферов. В силу того, что RAID5 гарантирует корректность работы случае выхода из строя не более одного стрипа в каждом страйпе, возникают два сценария при обработке запросов чтения.

1. Запрос чтения стрипов A2-A4



2. Запрос чтения стрипов A2-A4



$$A3 = A1 \text{ xor } A2 \text{ xor } A4 \text{ xor } A5 \text{ xor } A6 \text{ xor } P$$

Рис. 4: Два сценария запроса чтения.

3.3.1 Без битых стрипов

В этом сценарии базовые блочные устройства, которые напрямую задействованы в запросе не содержат битых стрипов. Необходимо разделить память, выделенную под чтение на отдельные буферы, после этого делегировать запросы чтения нужным устройствам. Обработка запроса закончится после того, как базовые устройства обработают запросы, делегированные им. Сценарий изображён на рис. 4 под номером один.

3.3.2 Один битый стрип

Во втором сценарии один из стрипов запроса находится в нерабочем состоянии, значит надо воспользоваться данными со всего страйпа, чтобы восстановить информацию. Для чтения данных с базовых дисков, неучаствующих в запросе, надо получить буферы для чтения данных. После этого необходимо отправить запросы чтения всем базовым устройствам, кроме того которое содержит битый стрип. И после завершения всех остаётся посчитать результат операции XOR на полученных данных. Сценарий представлен на рис. 4 под номером два.

3.4 Обработка запросов записи

Обработка запросов записи происходит несколько сложнее, чем обработка запросов чтения, а именно нужно гарантировать инвариант RAID5: в стрипе паритета хранится результат операции XOR с остальных стрипов страйпа. Для упрощения сценариев и сохранения инварианта, можно воспользоваться свойством операции XOR:

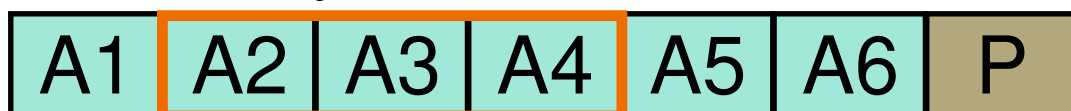
$$P \oplus A1 \oplus A1_{new} = (A1 \oplus A2 \oplus \dots \oplus A7) \oplus A1 \oplus A1_{new} = A1_{new} \oplus A2 \oplus \dots \oplus A7 = P_{new}$$

Для того чтобы гарантировать инвариант при старте RAID5 происходит инициализация RAID-массива, пересчитывающая паритет для каждого страйпа. Для этого необходимо считать данные со всего страйпа, выполнить операцию XOR и записать результат в стрип паритета. По

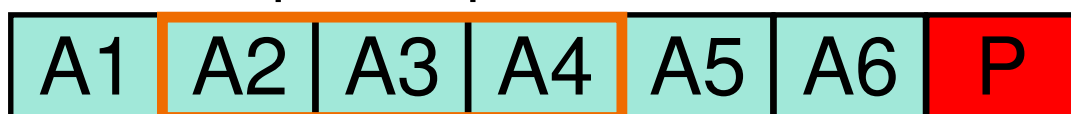
завершению инициализации состояние устройства корректно, и можно использовать стратегию записи “read-modify-write”, использующую свойство описанное выше, для которой не нужно перезаписывать весь страйп.

Для обработки запросов записи из функции `raid5_submit_rw_request` вызывается функция `raid5_submit_write_request`, в которой определяются сценарий записи и устанавливается соответствие между буферами и страйпами запроса. Все сценарии обобщаются до трёх случаев (рис. 5).

1. read-modify-write



2. битый страйп паритета



3. битый страйп запроса

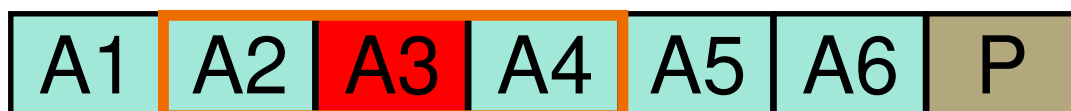


Рис. 5: Три сценария запроса записи.

3.4.1 Read-modify-write

Сценарий используется в случае, если среди страйпов запроса и страйпа паритета нет битых. В соответствии со стратегией “read-modify-write” устанавливается соответствие буфферов для хранения результатов запросов чтения со страйпами базовых дисков участвующих в запросе.

Происходит попытка чтения старых данных, после успешного завершения всех запросов чтения, вычисляется новый паритет, заменяются буферы, и отправляются запросы записи нужных стрипов из запроса к страйпу и стрипа паритета.

3.4.2 Битый стрип паритета

В этом случае невозможно перезаписать стрип паритета, а ,значит, и считать его не нужно. Необходимо только записать данные в запрашиваемые стрипы. Устанавливается соответствие буферов с запрашиваемыми стрипами. После отправляются запросы на запись нужных стрипов.

3.4.3 Битый стрип запроса

Информация, которую надо было бы записать в битый стрип, сохранится в стрипе паритета вместе с данными остальных стрипов, таким образом, необходимо пересчитать и перезаписать контрольно-восстановительную сумму а также записать остальные стрипы запроса. Посылаются запросы на чтение всех стрипов, кроме битого стрипа и стрипа паритета. Если все запросы отработали успешно, вычисляется новая контрольно-восстановительная сумма. После чего отправляются запросы записи изначально запрашиваемых стрипов и стрипа паритета.

3.5 Выделение памяти буферов запроса с глобальной адресацией

Для отправки запросов на устройства поддерживающие DMA, не подойдут обычные средства выделения памяти, такие как `malloc` или `calloc`, потому что они возвращают указатели с виртуальными адресами, которые станут не корректными при переключении контекста. Для решения этой проблемы в SPDK используются `spdk_dma_malloc` и аналоги, выделяющие память на `hugepages`, страницах оперативной памяти размером два мегабайта. При выделении памяти такими функциями возвращаются указатели с физическими адресами, однако вызов этих

функций сопряжён с накладными расходами. Если вызывать их во время обработки запроса, сильно увеличивается время выполнения одного. Для решения этой проблемы была использована абстракция каналов SPDK: `spdk_io_channel`.

3.5.1 RAID5 io channel

Канал представляет собой обёртку над данными ассоциированными с запросом, в ней можно хранить дополнительную информацию, например массив каналов базовых блочных устройств, а также такую обёртку можно переиспользовать для следующих запросов. Для использования этой функциональности при старте устройства необходимо вызвать функцию `spdk_io_device_register` и передать указатель на структуру ассоциированную с устройством, размер памяти для канала, а также функции создания и разрушения. Последние соответственно будут вызваны при создании канала и завершении его существования.

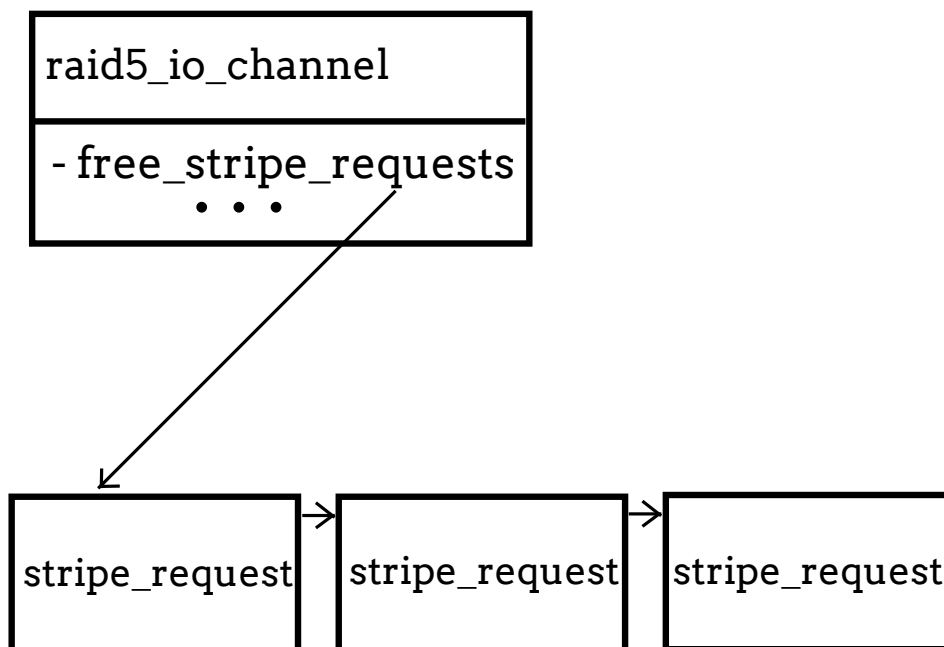


Рис. 6: `raid5_io_channel`.

3.6 Обработка запросов в многопоточном режиме

Для корректной работы RAID5 необходимо поддерживать инвариант в течении всего времени работы устройства. Это не накладывает ограничений на обработку запросов чтения в многопоточном режиме: они никак не изменяют контрольные суммы, однако запросы записи могут конкурировать в перезаписи стрипов паритета.

Представим ситуацию: два потока пытаются записать данные в один страйп. Между чтением данных первым потоком и записью новых данных им же, второй поток может считать данные, и на момент перезаписи стрипа паритета они уже не будут актуальны. Подобная ситуация приводит к нарушению инварианта.

Для решения подобного рода проблем, предлагается ввести блокировку страйпов на момент обработки запроса на запись. Для этого на старте RAID5 каждому страйпу сопоставляется пара из примитива синхронизации SPDK `spdk_spinlock` и флага `locked`. Сущность `spdk_spinlock` в купе с функциями `spdk_spin_lock` и `spdk_spin_unlock` является реализацией спин-блокировки, подразумевающей непродолжительное ожидание захвата. Чтобы заблокировать страйп, в начале обработки запроса записи с чтением стрипов происходит попытка взять блокировку и установить флаг, в случае неудачи страйп отправляется в очередь ожидания. В конце обработки запроса флаг сбрасывается без захвата спин-блокировки. Таким образом, в путь обработки запроса добавляются следующие функции:

- `raid5_try_lock_stripe` — для попытки установить флаг блокировки;
- `raid5_unlock_stripe` — для сброса флага;
- `raid5_write_unlock_and_complete` — обёртка над функцией завершения запроса с разблокировкой страйпа.

4 Тестирование

Тестирование полученной реализации проводится через Userspace block device driver. В тестах использовался следующий сценарий:

1. Bash-скрипт, создаёт блочные устройства, аллоцированные в пространстве пользователя и RAID5, с параметрами из конфигурационных файлов.
2. С помощью утилиты `fiio` [1] на них генерировалась нагрузка, описанное в конфигурационном файле теста с верификацией sha256 хэшей.
3. Удаляется одно из базовых устройств, этим шагом эмулируется выход из строя одного базового блочного устройства.
4. Снова генерируется нагрузка с верификацией.
5. RAID5 и базовые блочные устройства удаляются.

Конфигурационные файлы, передаваемые утилите `fiio` [1], используют разные сценарии, описывая нагрузку с запросами как по с размером равным стрипам RAID5, так и с размером больше одного стрипа. В ходе тестирования отправки синхронных запросов используется нагрузки двух видов: последовательная и случайная запись. Первый вид посылает запросы, адресуя непрерывные блоки по последовательным адресам на RAID5, второй же подразумевает выбор адреса случайным образом. При тестировании асинхронной отправки запросов используется последовательная запись, гарантируя конфликты у идущих друг за другом запросов. Кроме последовательной записи используется асинхронный движок отправки запросов `io_uring`, в то время как параметр `iodepth` принимает значения 4 или 16. Это позволяет смоделировать асинхронную нагрузку, засчёт постоянного числа отправленных запросов.

Удаление устройства и повторная генерация нагрузки позволяют проверить не только корректность работы в обычном режиме, но и с вышедшим из строя базовым устройством.

Заключение

В ходе выполнения работы были получены следующие результаты:

- Спроектирована система функций и callback-функций для регистрации запросов чтения и записи в RAID5.
- Реализовано выделение памяти буфферов запроса с глобальной адресацией.
- Поддержана обработка запросов в многопоточном режиме.
- Реализован RAID5 с обработкой запросов к страйпам³.
- Написаны интеграционные тесты для проверки корректности полученной реализации.

В дальнейшем планируется добавить поддержку восстановления данных при замене базового блочного устройства через интерфейс, предоставляемый SPDK.

³Полученная реализация

Список литературы

- [1] Fio documentation. — URL: <https://fio.readthedocs.io/en/latest/index.html> (дата обращения: 2023-12-21).
- [2] Hickmann Brian, Shook Kynan. Zfs and raid-z: The über-fs? // University of Wisconsin–Madison. — 2007.
- [3] Karol Latecki, Jaroslaw Chachulski. SPDK NVMe BDEV Performance Report Release 23.09. — 2023. — URL: https://ci.spdk.io/download/performance-reports/SPDK_nvme_bdev_perf_report_2309.pdf (дата обращения: 2023-12-21).
- [4] The SNIA Dictionary. — URL: <https://www.snia.org/education/online-dictionary> (дата обращения: 2023-12-21).
- [5] The SPDK Documentation, NVMe Driver. — URL: <https://spdk.io/doc/nvme.html> (дата обращения: 21 декабря 2023 г.).
- [6] ScalaRAID: Optimizing Linux Software RAID System for Next-Generation Storage / Shushu Yi, Yanning Yang, Yunxiao Tang et al. // Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems. — 2022. — P. 119–125.
- [7] Steven Briscoe, Thai Le. Accelerating your NVMe drives with SPDK. — 2017. — URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/accelerating-your-nvme-drives-with-spdk.html> (дата обращения: 2023-12-21).