Санкт-Петербургский государственный университет

Кафедра системного программирование

Группа 22.Б11-мм

Объединение запросов в RAID5 массиве

Елкин Леонид Николаевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
Старший преподаватель каф. СП, Я. А. Кириленко

Консультант:
Ведущий инженер-программист ООО «Швачер», А. И. Васенина

Санкт-Петербург
2023

Оглавление

Введение

Очень сложно в наше время переоценить значимость систем хранения данных, которые используются продвинутым пользователем практически каждый день. Блочное устройство является одной из технологий в данной отрасли, основанное на взаимодействии с устройством посредством обмена данными блоками. Однако простое блочное устройство довольно небезопасно и непроизводительно, поэтому их стали объединять в RAID(Redundant Array of Independent Disks) массивы [12, 17]. Вопрос их производительности стоит достаточно остро, так как их использование повсеместно, а мы хотим получить, как быструю, так и надежную систему.

RAID 5 и RAID 6 массивы – это широко используемые технологии за счет того, что дают хороший баланс отказоустойчивости и полезного места для хранения данных. Основной идеей этих массивов является хранение контрольно-восстановительных сумм, которые позволяют восстанавливать информацию при выходе диска из строя [14]. Контрольно-восстановительную сумму необходимо обновлять при каждом запросе на запись, чтобы она всегда содержала корректную избыточную информацию. Такой метод обеспечивает отказоустойчивость, однако уменьшает производительность из-за общего увеличения запросов на запись к RAID-массиву. Большое количество перезаписей повышает износ диска и тратит большее время, чем запись полезной нагрузки.

Одним из способов решения проблемы большого количества записей в контрольно-восстановительные суммы является объединение запросов. Если запросы образуют full-stripe, то есть переписывается весь ряд, именуемый страйпом, запросы можно объединить, а запись контрольной суммы произойдет один раз, что может значительно повысить производительность при большом количестве базовых блочных устройство в RAID-массиве. При обычной записи не full-stripe запроса считываются данные с диска, в который пишется информация, и с диска, в котором находится контрольно-вычислительная сумма, вычисляется новая контрольно-восстановительная сумма и затем записывается. Та-

кой метод называется RMW(read-modify-write) [18]. При объединении запроса не нужно считывать данные для вычисления новой контрольно-вычислительной суммы, что не только позволяет записать сумму единожды, но и не заставляет посылать запросы на чтение информации, находящийся в RAID-массиве.

В ядре linux взаимодействие пользователя с блочными устройствами происходит в пространстве ядра [1], а не в пространстве пользователя, однако есть фреймворк от компании Intel под названием SPDK(Storage Performance Development Kit), обеспечивающий взаимодействие с системами хранения данных не в пространстве ядра, а в пространстве пользователя. Такой подход, как заверяет компания, позволяет увеличить производительность, а также упрощает отладку и делает разработку удобней [9].

На данный момент в фреймворке SPDK не реализована функциональность объединения запросов, однако реализовано взаимодействие с другими RAID-массивами, а также ведется активная работа по реализации RAID 5 массива, что позволяет написать функциональность объединения запросов. В рамках данной работы планируется встроить реализацию объединения запросов в разрабатываемый RAID 5 массив, чтобы ускорить обработку запросов.

1 Постановка задачи

Целью работы является разработка алгоритма объединения последовательных запросов записи в RAID 5 массиве для уменьшения количества перезаписей за счет накопления изменений в оперативной памяти. Для достижения данной цели были поставлены следующие задачи:

1. Выполнить обзор фреймворка SPDK и его взаимодействие с пользовательскими запросами записи.

2. Предложить и реализовать алгоритм перехвата, отслеживания и объединения запросов записи.

3. Предложить и реализовать функциональность преждевременного объединения в случае отсутствия нужного количества запросов в течении долгого времени.

4. Протестировать производительность системы и целостность файлов, записанных с использованием данного алгоритма.

2 Обзор

‘’

2.1 Устройство RAID-массива

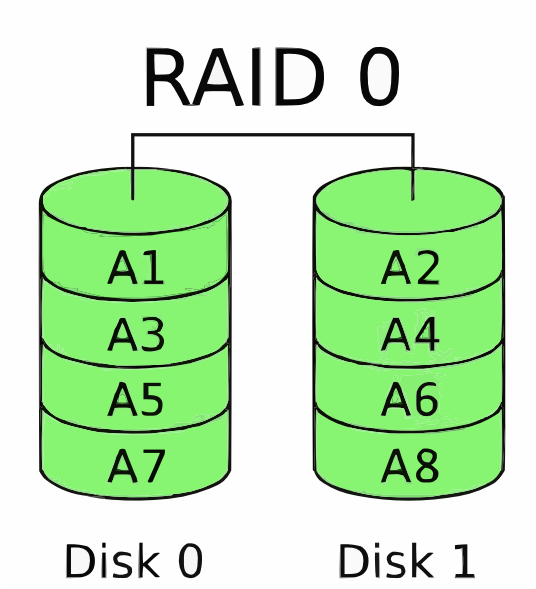
RAID-массив – это технология виртуализации хранилища данных, которая объединяет несколько компонентов физических дисков в один или несколько логических блоков с целью повышения отказоустойчивости, повышения производительности или и того, и другого. Данные распределяются по дискам одним из нескольких способов, называемых уровнями RAID, в зависимости от требуемого уровня избыточности данных и производительности.

RAID взаимодействует с базовыми блочными устройствами, из которых он состоит. Количество базовых блочных устройств, отведенных под избыточную информацию и общее количество базовых блочных устройств определяют уровень RAID-массива. Также количество дисков определяет размер страйпа(stripe). Один блок в одном из базовых блочных устройств называется стрипом(strip). Страйп представляет из себя один ряд из стрипов [17, 13].

Далее рассмотрим самые популярные уровни RAID-массивов, которые чаще всего используются на практике.

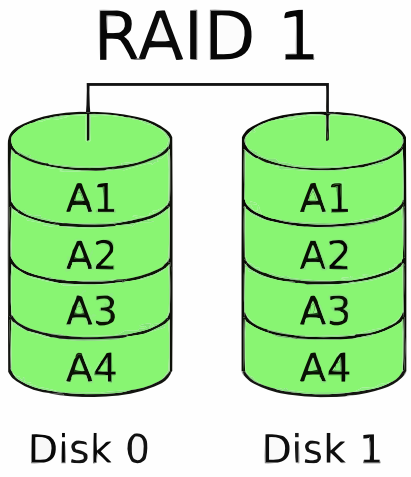
2.1.1 RAID 0

RAID 0 распределяет данные по двум или более дискам, используя страйпинг(striping). Страйпинг – это равномерное распределение информации по страйпу, что увеличивает производительность системы за счет параллельной записи в стрипы одного страйпа. RAID 0 не предоставляет отказоустойчивости и не содержит никакой избыточной информации, что делает его крайне уязвимым к потере информации при выходе хотя бы одного диска из строя. RAID 0 – это пример максимальной производительности при отсутствующей отказоустойчивости.



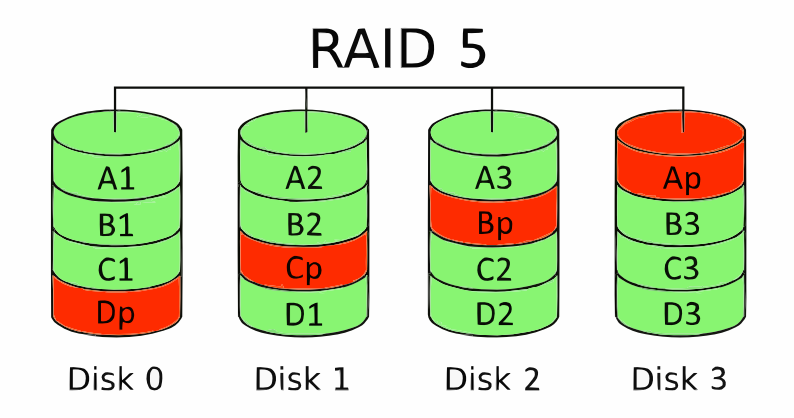
2.1.2 RAID 1

RAID 1 состоит из двух или более дисков и представляет собой данные и их точную копию, также именуемую зеркалом. Такая конфигурация не обеспечивает страйпинга или разделения дискового пространства на несколько дисков, поскольку данные зеркально отражаются на всех дисках, принадлежащих массиву, а размер массива равен размеру самого маленького диска. Такая схема полезна, когда надежность важнее, чем производительность записи или результирующая емкость хранилища данных.



2.1.3 RAID 5

RAID 5 состоит из трех или более дисков, из которых один отводится для хранения контрольно-восстановительной суммы, также именуемой паритетом. В каждом страйпе один стрип отводится под сумму, однако стрипы паритета не обязательно находятся на одном диске.



Стрипы паритета содержат избыточную информацию только о блоках страйпа, в котором они находятся, то есть паритет в первом страйпе на картинке отвечает только за блоки 0, 1 и 2.

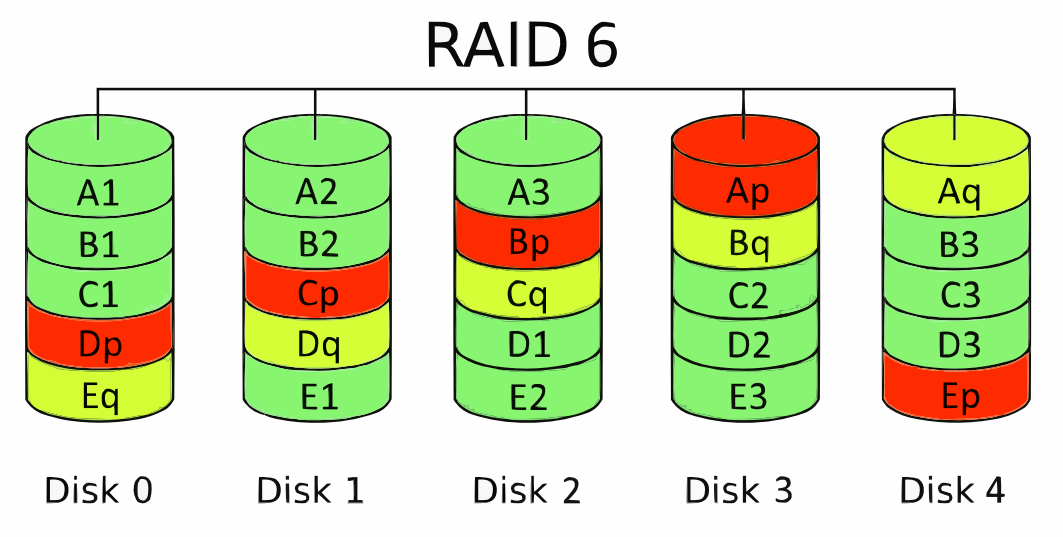
Сама контрольная сумма представляет собой операцию XOR всех остальных стрипов в страйпе. Так как операция XOR полностью коммутативна и самообратима, потеряв один диск информации, мы легко сможем восстановить информацию из утерянного диска, проведя XOR всех уцелевших данных в страйпе с блоком паритета. Если же из строя вышел диск, на котором находится стрип паритета, мы не теряем данные вовсе, а после того, как будет произведено восстановление диска, нужно пересчитать и записать контрольно-вычислительную сумму.

Таким образом RAID 5 предоставляет хороший баланс между отказоустойчивостью, производительностью и количеству полезной информации по сравнению с избыточной.

2.1.4 RAID 6

RAID 6 расширяет RAID 5 за счет добавления еще одного блока паритета. Таким образом, он использует страйпинг с двумя стрипами паритета вместо одного, как было в RAID 5.

Как и в RAID 5, существует несколько схем дисковых массивов RAID 6 в зависимости от направления и расположения стрипов паритета относительно стрипов данных и того, записывается ли первый стрип данных последующего страйпа на тот же диск, что и последний стрип паритета предыдущего страйпа. На рисунке 4 приведен пример



Для вычисления паритета в RAID 6, помимо операции XOR, используются коды Рида-Соломона. Вычисление кодов, в отличие от вычисления XOR, нагружает процессор, так как оно содержит полиномиальное умножение, что снижает производительность записи в RAID 6-массив.

RAID 6, в отличие от RAID 5, предоставляет отказоустойчивость в два диска, однако уступает в производительности и занимаемому месту относительно полезной нагрузки.

2.2 Обзор существующих аналогов

Для исследования существующих алгоритмов по объединению запросов записи были проанализированы реализации планировщиков ввода-вывода в Linux. Объединение запросов обычно является одним из действий, выполняемых планировщиками ввода-вывода Linux для минимизации запросов при работе с жесткими дисками. Наиболее известными являются CFQ(Completely fair queueing) и Deadline [15].

2.2.1 CFQ

Планировщик ввода-вывода CFQ распределяет входящие запросы ввода-вывода по определенным очередям на основе процесса, инициировавшего запрос. Внутри каждой очереди запросы объединяются со смежными запросами и сортируются при записи. Таким образом, очереди сохраняются отсортированными по секторам. Затем планировщик ввода-вывода CFQ обслуживает очереди циклически, отбирая настраиваемое количество запросов из каждой очереди, прежде чем переходить к следующей. Такой подход обеспечивает справедливость на уровне каждого процесса, гарантируя, что каждый процесс получает справедливую долю пропускной способности диска.

Также в CFQ реализована функциональность slice idle. Это числовое значение, которое показывает как долго будет простаивать планировщик, ожидая новый запрос в одной из очередей CFQ. Slice idle может принимать нулевое значение, это означает, что в работе очередей не будет простоев. Такой подход может как повысить пропускную способность, так и уменьшить. Обычно нулевой slice idle выставляют на более быстрых устройствах хранения данных, таких как несколько дисков SATA/SAS в конфигурации аппаратного RAID. Недостатком является то, что изоляция, обеспечиваемая при записи, также снижается, и понятие приоритета ввода-вывода становится слабее, а также уменьшается производительность некоторых устройств хранения данных [2].

2.2.2 Linus Elevator

Прежде чем рассмотреть Deadline, стоит понять, как работает планировщик ввода-вывода Linus Elevator, так как Deadline в основном работает аналогично за исключением некоторых деталей [11].

Linus Elevator состоит из очереди запросов, отсортированной по физическому расположению секторов. Когда запрос добавляется в очередь, он сначала сравнивается с каждым другим ожидающим запросом, чтобы определить, является ли он возможным кандидатом на объединение. Если новый запрос продолжает существующий запрос, он объеди-

няется спереди, выполняя слияние, именуемое front merge. И наоборот, если новый запрос предшествует существующему запросу, он объединяется сзади, выполняя back merge. Из-за способа размещения файлов и операций ввода-вывода, выполняемых при типичной рабочей нагрузке, front merge встречается очень редко по сравнению с back merge. Тем не менее, Linus Elevator проверяет и выполняет оба типа слияния.

Если попытка слияния завершается неудачей, выполняется поиск возможной точки вставки в очередь. Это такое место в очереди, где новый запрос помещается по секторам между существующими запросами. Если такое место найдено, новый запрос вставляется туда. Если подходящее местоположение не найдено, запрос добавляется в конец очереди.

Кроме того, если в очереди обнаружен существующий запрос, возраст которого превышает заданный порог, новый запрос добавляется в конец очереди, даже если его можно отсортировать относительно существующих запросов. Такой подход призван предотвратить бесконечное "голодание" запросов к секторам, расположенным в других местах диска.

К сожалению, эта проверка "возраста" не очень эффективна. Она не обеспечивает никакой гарантии обслуживания запросов в какой-то промежуток времени, а просто останавливает сортировку запросов после заданной задержки. Это улучшает задержку, но все равно может привести к нехватке запросов. Поток запросов к одной и той же области диска может привести к тому, что другие удаленные запросы никогда не будут обслужены.

2.2.3 Deadline

Планировщик ввода-вывода Deadline призван предотвратить бесконечное "голодание" запросов, которое было в Linus Elevator. В планировщике ввода-вывода Deadline каждый запрос связан со своим временем истечения срока действия. По умолчанию время истечения составляет 500 миллисекунд для запросов на чтение и 5 секунд запросов на запись [11].

Планировщик ввода-вывода Deadline работает аналогично Linus Elevator в том смысле, что он поддерживает очередь запросов, отсортированную по физическому расположению на диске. Он называет эту очередь отсортированной очередью. Когда новый запрос отправляется в отсортированную очередь, планировщик ввода-вывода Deadline выполняет слияние и вставку подобно Linus Elevator.

Однако Deadline также вставляет запрос во вторую очередь, которая зависит от типа запроса. Запросы на чтение и запись сортируются по времени в отдельные очереди FIFO(First In First Out). Следовательно, новые запросы всегда добавляются в конец очереди. При нормальной работе планировщик ввода-вывода по крайнему сроку извлекает запросы из заголовка отсортированной очереди в очередь отправки. Очередь отправки затем передается на диск. Это приводит к минимальному количеству запросов.

Если срок действия запроса, находящегося в начале очереди записи или чтения, истекает, планировщик ввода-вывода Deadline начинает обслуживать запросы из очереди. Таким образом, он пытается гарантировать, что ни один запрос не остается невыполненным дольше срока его истечения.

Стоит заметить, что планировщик ввода-вывода Deadline не дает никаких строгих гарантий в отношении задержки запроса. Однако он способен, как правило, фиксировать запросы по истечении срока их действия или до него. Это предотвращает "голодание" запросов. Поскольку запросам на чтение присваивается значительно меньшее значение истечения срока действия, чем запросам на запись, планировщик ввода-вывода Deadline также обеспечивает, что запросы на запись не перегружают запросы на чтение. Такое предпочтение запросам на чтение обеспечивает минимальную задержку чтения.

2.2.4 RAID-Z

Еще одним подходом к уменьшению объема записи сумм паритета является расположение данных, настраиваемое файловой системой. Одним из способов является комбинация файловой системы ZFS и RAID,

реализованная в RAID-Z. RAID-Z не имеет фиксированного размера страйпа, как в классических RAID-массивах, в которых размер страйпа определяется количеством дисков при создании. Когда приходит запрос на запись, ZFS сама подбирает страйп с идеально подходящей длиной равной длине пришедшего запроса [16].

Таким образом, нет необходимости объединять запрос, если он уже по приходе является объединенным для соответствующего страйпа. Такой подход является намного более эффективным на больших объемах данных по сравнению с RMW в классическом RAID-массиве, однако на маленьких объемах данных тратится большое количество времени на запись метаданных citeRAIDz1.

В контрольно-восстановительных суммах содержится ссылка на тот объем данных, к которым они относятся. Таким образом данные и сумма могут не находиться непосредственно рядом в памяти. Технология copy-on-write активно этим пользуется. Она при изменении уже записанных данных копирует их в другое место в памяти, обновляет контрольно-восстановительную сумму и атомарно меняет ссылку в ней на новый фрагмент в памяти. Старые и новые данные при этом могут находиться в одном страйпе, что позволяет избежать такой проблемы классических RAID-массивов как "write hole", когда данные успели записаться в стрип полезной нагрузки, но не успели записаться в контрольно-восстановительную сумму в ходе, например, сбоя в электричестве, что делает информацию в RAID-массиве некорректной. Однако такая реализация влечет за собой фрагментацию памяти, что может уменьшить фактическое доступное место в RAID-Z массиве [3].

2.3 Устройство SPDK

SPDK предоставляет набор инструментов и библиотек для написания высокопроизводительных масштабируемых приложений хранения данных в пространстве пользователя. Высокая производительность, как заверяют создатели, достигается по следующим причинам [9]:

- Перемещение всех необходимых драйверов в пространство пользователя, что позволяет уменьшить количество системных вызовов и обеспечивает доступ из приложения без копирования.

- Опрос оборудования на предмет завершения вместо использования прерываний.

- Избегание блокировок на пути ввода-вывода с помощью передачи сообщений.

Основой SPDK является асинхронный NVMe драйвер пользовательского пространства, работающий в режиме опроса. Это обеспечивает параллельный, zero-copy доступ к SSD из приложения пользовательского пространства.

SPDK делится на серверную и клиентскую части [8]. Для общения между серверной и клиентской частями SPDK предоставляет набор скриптов, написанных на python, которые могут выполнять сценарии, описанные в JSON-RPC [6, 10]. JSON-RPC представляет из себя протокол RPC(Remote Procedure Call), в котором описаны методы, структуры данных и правила их обработки. Такой подход позволяет запускать SPDK удаленно или через стороннее программное обеспечение.

Основная реализации SPDK со стороне сервера делится на две части: `lib` и `module`. `lib` состоит из набора библиотек, которые описывают ту или иную сущность взаимодействия с SPDK. В рамках данной работы нас интересует библиотека `lib/bdev`, которая реализует абстракции взаимодействия с блочным устройством. В свою очередь в `module` находятся реализации устройств, использующих `lib`. В `module` также присутствуют реализации блочных устройств, использующих `lib/bdev`, они находятся в `module/bdev`. В ней представлены реализации таких блочных устройств как: null, RAID 0 и другие, а также написаны заготовки для реализации других блочных устройств [4].

В SPDK также содержатся заголовочные файлы общедоступного интерфейса, находящиеся в

`include/spdk`. Они предоставляют интерфейс к функциональности, написанной в `lib`. Так, например, для написания нового блочного устройства необходимо воспользоваться `bdev.h` и `bdev_module.h`, в которых содержатся структуры, описывающие блочные устройства, и функции для взаимодействия с ними. Также в `include/spdk` описаны вспомогательные библиотеки, описывающие, например, различные структуры данных [7].

SPDK имитирует многопоточность путем общедоступного интерфейса `thread.h`. Любой модуль может инициализировать абстракцию потоков и предоставлять обратные вызовы для реализации callback функций, необходимых библиотекам SPDK. SPDK часто назначает глобальные данные одному потоку. Когда другие потоки хотят получить доступ к данным, они передают сообщение потоку-владельцу, чтобы тот выполнил операцию от их имени.

Альтернативой прерываний в SPDK выступает функция `poller` для каждого отдельного модуля. Она представляет из себя функцию, которая бесконечно выполняется в отдельном потоке. Такой подход позволяет асинхронно реагировать на события и выполнять соответствующие действия, прописанные в функции [5].

3 Реализация

3.1 Перехват запроса

Реализация RAID 5 в SPDK написана в файле `module/bdev/raid/raid5.c`. В SPDK RAID 5 запрос попадает в функцию `raid5_submit_rw_request`. Структура `raid_bdev_io` описывает запрос к RAID-массиву, а именно тип запроса: запись или чтение, данные, находящиеся в нем и так далее. В приведенной выше функции определяется тип запроса, и на основе этого запрос передается в соответствующие функции его выполнения. В пределах данной задачи взаимодействие происходит с запросами на запись.

Перехват запроса происходит следующим образом. Когда приходит запрос на запись, он пересылается в отдельную функцию, находящуюся в другом файле `raid_merge_requests.c`, подключенную с помощью заголовочного файла в `raid.c`. Написанная функция `raid_request_catch` впоследствии взаимодействует с полученным запросом.

Полученные запросы необходимо где-то хранить, пока не наберется full-stripe. Для этого были выбраны следующие следующие структуры данных. Запросы, поступающие к одному страйпу RAID-массива, хранятся в Rank-Balanced tree, разделенные по стрипам. В свою очередь, деревья хранятся в хеш-таблице по соответствующим страйпам.

3.1.1 Сохранение запросов в Rank-Balanced tree

В Rank-Balanced tree запросы сортируются по адресам стрипов, к которым направлен запрос на запись. Такой подход позволяет легко обходить дерево и объединять запросы. Реализация Rank-Balanced tree уже была написана в SPDK и находится в `include/spdk/tree.h`, поэтому было решено использовать ее. Для взаимодействия с деревом было создано две новые структуры: узел дерева и само дерево.

Узел — это структура, которая содержит сравнимый тип данных, чтобы сортировать узлы в дереве, структуру, которая хранится в дереве, и ссылки на двух детей и родителя данного узла.

```
struct raid_write_request {
  uint32_t addr;
  RB_ENTRY(raid_write_request) link;
  struct raid_bdev_io *bdev_io;
}
```

В данной структуре поле `uint32_t addr` является сравнимым типом данных и отвечает за сортировку запросов по адресам. Макрос `RB_ENTRY(raid_write_request) link` создает структуру, содержащую ссылки на родителя и двух детей. Структура `struct raid_bdev_io *bdev_io` содержит хранимый запрос.

Стоит отметить, что структура `raid_bdev_io` описывает запрос к абстрактному RAID-массиву. Каждый отдельный RAID-массив может содержать свою структуру запроса, в которой одним из полей будет структура `raid_bdev_io`. В дереве хранятся структуры `raid_bdev_io`, чтобы написанную функциональность можно было переиспользовать в других RAID-массивах, в частности в RAID 6.

Далее необходимо создать структуру самого дерева. Это не является обязательным пунктом, так как можно представлять корень дерева как узел без родителя, однако такой подход облегчает отслеживание размера дерева и помогает оперировать несколькими деревьями, что облегчает взаимодействие с деревом в хеш-таблице.

```
struct raid_request_tree {
  RB_HEAD(raid_addr_tree, raid_write_request) tree;
  uint64_t size;
}
```

Макрос `RB_HEAD(raid_addr_tree, raid_write_request) tree` создает структуру с единственным полем, в которой описан корень дерева. Поле `uint64_t size` отвечает за размер дерева, то есть за количество стрипов одного страйпа, к которым поступили запросы на запись.

3.1.2 Сохранение деревьев в хеш-таблицу

Запросы могут поступать непоследовательно и не к одному страйпу. Поэтому деревья по разным страйпам тоже необходимо хранить. Для

этого была выбрана структура хранения данных хеш-таблица. Ее реализации не было в SPDK, поэтому было принято решение написать ее.

Хеш-таблица предоставляет функции ее создания и удаления, получения значения по ключу, добавления и удаление элемента по ключу, а также специальную функцию итерации по таблице. Ключом в хеш-таблице является адрес начала страйпа, а значением — дерево, соответствующее данному страйпу.

Когда набирается full-stripe, запросы из дерева объединяются в один, а дерево удаляется из хеш-таблицы. Такое решение было принято потому, что страйпов может быть очень много, а хранить столько деревьев затратно по памяти. Конечно, если поступит full-stripe запросов, произойдет объединение и дерево удалится, может случиться ситуация, что запросы продолжат поступать к данному страйпу и придется пересоздавать дерево, которое только что было создано, однако такая ситуация крайне редка из соображений того, что информация только что записанная вероятно будет удалена нескоро.

Хеш-таблица имеет переменный размер. Это сделано для того, чтобы в случае маленькой нагрузки не занимать много памяти изначально. При заполненности наполовину хеш-таблица увеличится вдвое. Такая стратегия была выбрана во избежание коллизии, и, несмотря на то что обработка коллизии предусмотрена в данной реализации хеш-таблицы, она занимает время.

3.2 Объединение запросов

Для того, чтобы создать запрос в SPDK, необходимо его зарегистрировать. Структура `spdk_bdev_io` описывает запрос к блочному устройству. Задача состоит в том, чтобы создать и зарегистрировать новую структуру `spdk_bdev_io`, в которой будут данные всех объединенных запросов, а также подключить к ней с помощью контекста структуру `raid_bdev_io`.

Контекст — это массив нулевого размера в конце структуры. Это

позволяет присвоить контексту другую структуру. Так, например, структура `raid_bdev_io` может являться контекстом к структуре `spdk_bdev_io`. Такой подход позволяет специализировать структуру `spdk_bdev_io` на конкретном блочном устройстве и знать, что запрос направлен именно к нему. Также в SPDK реализованы функции получения информации из контекста, так, например, имея доступ только к структуре `raid_bdev_io`, можно с ее помощью получить доступ к `spdk_bdev_io`.

3.2.1 Регистрация дескриптора

Для начала необходимо зарегистрировать дескриптор. Дескриптор предназначен для контроля параллельных запросов на запись и чтение. Он дает разрешение и выделяет каналы, по которым впоследствии будут отправляться запросы. Дескриптор необходим, так как координация всех запросов чтения и записи должна выполняться структурой более высокого уровня. Он описывается структурой `spdk_bdev_desc` У одного блочного устройства может быть несколько дескрипторов. Дескриптор оперирует такими терминами как канал и потоки. Он выделяет потоку канал, по которому он будет общаться с блочным устройством и посылать на него запросы.

Регистрация дескриптора происходит с помощью следующей функции.

```
int spdk_bdev_open_ext(const char *bdev_name, bool write,
    spdk_bdev_event_cb_t event_cb, void *event_ctx,
    struct spdk_bdev_desc **_desc)
```

Функция принимает следующие параметры. `const char *bdev_name` — это имя блочного устройства, для которого нужно зарегистрировать дескриптор. `bool write` отвечает за то, можно ли через этот дескриптор осуществлять запросы на запись или нет. `true` — можно, `false` — нельзя. `spdk_bdev_event_cb_t event_cb` — это callback функция, которая описывает указания, которые нужно выполнить при получении асинхронного события. Событием может быть какое-либо взаимодействие с блочным устройством, например, его удаление. `void *event_ctx` — это аргументы для callback функции. `struct spdk_bdev_desc **_desc` — это указатель на

указатель дескриптора, с помощью него функция передает указатель на дескриптор.

3.2.2 Регистрация канала ввода-вывода

После регистрации дескриптора необходимо выделить канал запросов ввода-вывода, с помощью которого можно общаться с блочным устройством. Канал описывается структурой `spdk_io_channel`. Для этого в SPDK существует функция `spdk_bdev_get_io_channel`. Она принимает один дескриптор и выделяет канал.

Также необходимо получить обертку для `spdk_io_channel` в виде структуры `spdk_bdev_channel`. Эта структура состоит из одного поля — указателя на `spdk_io_channel`. Такая обертка необходима, чтобы уточнить, что общение происходит именно с блочным устройством. `spdk_bdev_channel` создается с помощью функции получения из контекста `spdk_io_channel_get_ctx`. Она принимает `spdk_io_channel` выполняет арифметику с указателями и возвращает указатель на структуру `spdk_bdev_channel`.

3.2.3 Создание и регистрация объединенного запроса

После регистрации дескриптора и канала, с помощью функции `bdev_channel_get_io` необходимо получить структуру `spdk_bdev_io`, которая будет описывать новый объединенный запрос.

Одними из важных полей в структуре `spdk_bdev_io` являются вектора ввода-вывода **`struct`** `iovec *iovs` и **`int`** `iovcnt`. Они описывают блоки данных, которые находятся в запросе на запись, и их количество соответственно. Идея в том, чтобы объединить массивы векторов ввода-вывода запросов, находящихся в дереве по страйпу, и обновить их количество.

После вычисления векторов ввода-вывода необходимо заполнить поля структуры `spdk_io_channel` и подтвердить регистрацию.

```
bdev_io->internal.ch = channel;
bdev_io->internal.desc = desc;
bdev_io->type = SPDK_BDEV_IO_TYPE_WRITE;
bdev_io->u.bdev.iovs = iovs;
```

```
bdev_io->u.bdev.iovcnt = iovcnt;
bdev_io->u.bdev.md_buf = NULL;
bdev_io->u.bdev.num_blocks = num_blocks;
bdev_io->u.bdev.offset_blocks = offset_blocks;
bdev_io->u.bdev.memory_domain = NULL;
bdev_io->u.bdev.memory_domain_ctx = NULL;
bdev_io->u.bdev.accel_sequence = NULL;
bdev_io_init(bdev_io, bdev, spdk_bdev_io_complete, NULL);
bdev_io_submit(bdev_io);
```

Первые пять полей, за исключением `type`, были подробно рассмотрены ранее. `type` — это число, которое определяет какой запрос регистрируется. Поля `md_buf`, `memory_domain`, `memory_domain_ctx` и `accel_sequence` не являются необходимыми в рамках данной задачи, поэтому им присваивается `NULL`, и останавливаться на них подробно не будем. Поле `num_blocks` описывает количество блоков, которые занимает информация, записываемая на блочное устройство. Вычисляется посредством деления длины всех векторов ввода-вывода на размер блока. Размер блока фиксирован и определяется при создании блочного устройства, обычно он равен 512б или 4Кб. Поле `offset_blocks` — это расстояние в блоках от начала структуры запроса до начала блоков данных. В случае объединенного запроса `offset_blocks` будет равен `offset_blocks` самого первого запроса в дереве, то есть с наименьшим адресом.

Функция `bdev_io_init` инициализирует запрос, заполняя оставшиеся поля структуры самостоятельно. Ее параметрами являются сам запрос, блочное устройство, которому запрос адресован, а также callback функция, которая вызывается, когда запрос выполнен, и параметры к ней.

Функция `bdev_io_submit` регистрирует запрос, после этого его можно отправлять на блочное устройство.

3.2.4 Подключение контекста к запросу

Запрос уже зарегистрирован, но чтобы адресовать его RAID-массиву, необходимо подключить структуру `raid_bdev_io` как контекст и заполнить поля в ней.

```
raid_io = (struct raid_bdev_io *)bdev_io->driver_ctx;
```

```
raid_io->raid_bdev = bdev_io->bdev->ctxt;
raid_io->raid_ch = spdk_io_channel_get_ctx(ch);
raid_io->base_bdev_io_remaining = 0;
raid_io->base_bdev_io_submitted = 0;
raid_io->base_bdev_io_status = SPDK_BDEV_IO_STATUS_SUCCESS;
```

Первые две строчки подключают контекст к структурам `spdk_bdev_io` и `spdk_bdev` соответственно. Полю `raid_ch` присваивается тот же канал, который мы присваивали `spdk_bdev_io`, потому что `raid_bdev_io` специализирует `spdk_bdev_io`, а не является новым запросом. Последние три строчки описывают текущее состояние запросов. Функции, в которых происходят взаимодействия с `raid_bdev_io` будут изменять эти значения.

3.3 Внедрение в RAID 5

Внедрение происходит в функцию `raid5_submit_rw_request`. В ней осуществляется перехват запроса и его перенаправление в функцию `raid_request_catch`, описанную ранее.

3.3.1 Обработка состояний запросов

В зависимости от заполненности дерева функция возвращает одно из числовых значений, описывающих состояние накопленных запросов.

```
enum raid_request_merge_status {
    RAID_REQUEST_MERGE_STATUS_COMPLETE = 0,
    RAID_REQUEST_MERGE_STATUS_WAITING_FOR_REQUESTS = 1,
    RAID_REQUEST_MERGE_STATUS_FAILED = -1,
}
```

Если после поступления запроса дерево становится заполненным, то есть образуется full-stripe, происходит объединение запросов, и уже объединенный запрос поступает обратно в функцию `raid5_submit_rw_request`, а функция объединение возвращает `RAID_REQUEST_MERGE_STATUS_COMPLETE`.

Если после поступления запроса дерево все еще не заполнено, функция возвращает `RAID_REQUEST_MERGE_STATUS_WAITING_FOR_REQUESTS`. Это означает, что ожидается поступление новых запросов.

В случае возникновения ошибки, функция возвращает `RAID_REQUEST_MERGE_STATUS_FAILED` и принудительно завершивает обработку всех поступивших запросов с сообщением об ошибке.

3.3.2 Completion функции

В SPDK после завершения выполнения запроса необходимо передать его в completion функцию с статусом выполнения. После объединения запросов, нужно передать в completion функции все запросы, из которых он состоит при этом сам объединенный запрос не нужно передавать в completion функцию, потому что его пользователь не отправлял.

Был выбран следующий подход. В случае, если объединенный запрос завершается с успехом, все запросы, из которых он состоял, также завершаются с успехом. Аналогично, если объединенный запрос завершается неудачей, все запросы, из которых он состоял, также завершаются неудачей.

Стоит отметить, что если произошла ошибка на этапе заполнения дерева запросам, все запросы, находящиеся в нем автоматически считаются завершенными с ошибкой. Такой подход обусловлен сложностью и большой временной затратностью вычисления запроса в дереве, который вызвал ошибку.

4 План тестирования

Тестирование написанного планируется проводить в весеннем семестре, после реализации всей задуманной функциональности, потому что к моменту промежуточного отчета не удалось полностью внедрить написанный код в RAID 5.

4.1 Условия тестирования

SPDK поддерживает только операционные системы на базе ядра linux, поэтому тестирование планируется проводить на дистрибутиве Ubuntu server последней версии.

Интеграционное тестирование будет произведено с помощью консольных утилит `dd` и `fio`.

Утилита `dd` позволяет производить тривиальную запись последовательно идущих блоков, что покажет работу реализованной функциональности в тривиальных случаях.

С помощью утилиты `fio` будут произведены более специализированные тесты с непоследовательной нагрузкой и проверкой записанных данных. Данная утилита имеет JSON файл, в котором присутствует гибкая настройка запросов ввода-вывода. Ее использование планируется, чтобы оценить производительность в различных нетривиальных случаях.

Модульное тестирование будет произведено в фреймворке SPDK для оценки работы каждого отдельного этапа в объединении запросовю

4.2 Исследовательские вопросы

В ходе тестирования будут выставлены следующие исследовательские вопросы:

- Какой выигрыш в производительности дает представленный алгоритм объединения запросов при последовательной внутри страйпа нагрузке?

- Насколько критична потеря производительности при редкой непоследовательной внутри страйпа нагрузке?

Заключение

В рамках проделанной работы на данный момент были получены следующие результаты

- Был выполнен обзор SPDK, изучен код, реализующий взаимодействие пользователя с блочными устройствами и запросами записи в частности.
- Был предложен и реализован алгоритм для перехвата, отслеживания и объединения запросов записи.
- Была начата работа по реализации преждевременного объединения в случае отсутствия нужного количества запросов в течении долгого времени. Планируется завершить данную функциональность в весеннем семестре.
- Был составлен план тестирования написанной функциональности. Планируется протестировать в весеннем семестре.

В весеннем семестре планируется завершить внедрение функциональности объединения запросов в RAID 5, дописать функциональность преждевременного объединения запросов, а также протестировать производительность и корректность всего кода.

Список литературы

[1] Archives The Linux Kernel. — Block Device Drivers. — URL: https://linux-kernel-labs.github.io/refs/heads/master/_sources/labs/block_device_drivers.rst.txt (дата обращения: 21 декабря 2023 г.).

[2] Archives The Linux Kernel. — Documentation of CFQ (Complete Fairness Queueing). — URL: https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt (дата обращения: 21 декабря 2023 г.).

[3] Corporation QueTek Consulting. ZFS and RAID-Z recoverability and performance. — URL: https://www.quetek.com/zfsandraidz.htm (дата обращения: 21 декабря 2023 г.).

[4] Documentation SPDK. — Block Device Layer Programming Guide. — URL: https://spdk.io/doc/bdev_pg.html (дата обращения: 21 декабря 2023 г.).

[5] Documentation SPDK. — Event Framework. — URL: https://spdk.io/doc/event.html (дата обращения: 21 декабря 2023 г.).

[6] Documentation SPDK. — JSON RPC. — URL: https://spdk.io/doc/jsonrpc.html (дата обращения: 21 декабря 2023 г.).

[7] Documentation SPDK. — Message Passing and Concurrency. — URL: https://spdk.io/doc/concurrency.html (дата обращения: 21 декабря 2023 г.).

[8] Documentation SPDK. — SPDK Strctural overview. — URL: https://spdk.io/doc/overview.html (дата обращения: 21 декабря 2023 г.).

[9] Documentation SPDK. — What is SPDK. — URL: https://spdk.io/doc/about.html (дата обращения: 21 декабря 2023 г.).

[10] Group JSON-RPC Working. — JSON-RPC 2.0 Specification. — URL: https://www.jsonrpc.org/specification (дата обращения: 21 декабря 2023 г.).

[11] Love Robert. Linux Kernel Development Second Edition. — URL: https://litux.nl/mirror/kerneldevelopment/0672327201/ch13lev1sec5.html (дата обращения: 21 декабря 2023 г.).

[12] RAID setup. — URL: https://raid.wiki.kernel.org/index.php/RAID_setup#RAID-4.2F5.2F6 (дата обращения: 21 декабря 2023 г.).

[13] SNIA. — URL: https://www.snia.org (дата обращения: 21 декабря 2023 г.).

[14] Syngress. Designing SQL Server 2000 Databases. — URL: https://www.sciencedirect.com/topics/computer-science/parity-information (дата обращения: 21 декабря 2023 г.).

[15] Vasenina Anna. Selective block I/O buffering for SSD-based RAID to reduce parity update overhead // Fifth Conference on Software Engineering and Information Management (SEIM-2020).

[16] What is RAIDZ. — URL: https://www.raidz-calculator.com/what-is-raidz.aspx (дата обращения: 21 декабря 2023 г.).

[17] Wikipedia. — RAID. — URL: https://en.wikipedia.org/wiki/RAID (дата обращения: 21 декабря 2023 г.).

[18] Wikipedia. — Read–modify–write. — URL: https://en.wikipedia.org/wiki/Read-modify-write (дата обращения: 21 декабря 2023 г.).