

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

Объединение запросов в RAID массивах

Елкин Леонид Николаевич

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2024

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Устройство RAID-массива	6
2.2. Обзор существующих аналогов	7
2.3. Устройство SPDK	10
3. Реализация	12
3.1. Перехват запроса	12
3.2. Объединение запросов	14
4. Тестирование	16
4.1. Условия тестирования	16
4.2. Параметры тестирования	16
4.3. Сценарий тестирования	17
4.4. Результаты тестирования	17
Заключение	19
Список литературы	20

Введение

Очень сложно в наше время переоценить значимость систем хранения данных, которые используются продвинутым пользователем практически каждый день. Блочное устройство является одной из основополагающих технологий в данной отрасли. Оно основано на взаимодействии с устройством посредством обмена данных блоками, а не символами [21]. Однако, простое одиночное блочное устройство небезопасно и непроизводительно, поэтому их стали объединять в RAID (Redundant Array of Independent Disks) массивы [13, 19]. Вопрос производительности RAID-массивов 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, которые 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 [15]. 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. Каждое базовое блочное устройство делится на стрипы (strips), которые могут состоять из одного или нескольких блоков. Именно к ним обращаются запросы на чтение или запись. Несколько стрипов, которые располагаются по одному адресу, но на разных блочных устройствах, образуя тем самым ряд, именуют страйпом (stripe). Если запросы на запись образуют full-stripe, то есть запись происходит в целый ряд, за исключением блока или блоков контрольной суммы, тогда запросы можно объединить, а запись контрольной суммы произойдет один раз, что

может значительно повысить производительность при большом количестве базовых блочных устройств в RAID-массиве. [16, 12] При обычной записи 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 вновь записывается. Такой метод называется RMW(read-modify-write) [20]. 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 напрямую [1], а не в пространстве пользователя, однако фреймворк SPDK(Storage Performance Development Kit) от компании Intel обеспечивает 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 реализовать интерфейс объединения запросов, который можно было бы легко встроить в любой RAID-массив, чтобы ускорить обработку запросов там где это возможно.

1 Постановка задачи

Целью работы является разработка алгоритма объединения последовательных запросов записи в RAID-массивах 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 полиморфный относительно типа RAID-массива алгоритм перехвата, отслеживания и объединения последовательных запросов записи.
3. Предложить и реализовать функциональность преждевременного объединения в случае отсутствия нужного количества запросов в течении долгого времени.
4. Внедрить написанную функциональность в SPDK RAID 5-массив.
5. 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. [19, 14].

Далее рассмотрим уровни RAID-массивов, с которыми мы столкнемся при непосредственном объединении запросов.

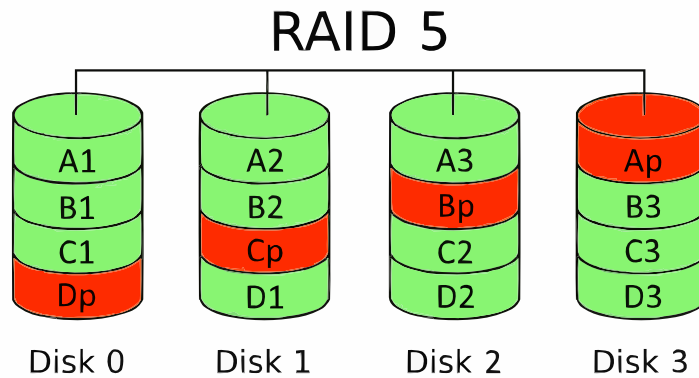
2.1.1 RAID 0 и RAID 1

RAID 0 распределяет данные по двум или более дискам, равномерно распределяя информацию по страйпу. За счет этого увеличивается производительность, не предоставляя отказоустойчивость.

RAID 1 состоит из двух или более дисков и представляет собой данные и их точную копию. Является противоположностью RAID 0 по идее, предоставляя хорошую отказоустойчивость, но меньшую производительность.

2.1.2 RAID 5 и RAID 6

RAID 5 состоит из трех или более дисков, из которых один отводится для хранения контрольно-восстановительной суммы, также именуемой парити(parity). 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 A1, A2 и A3.

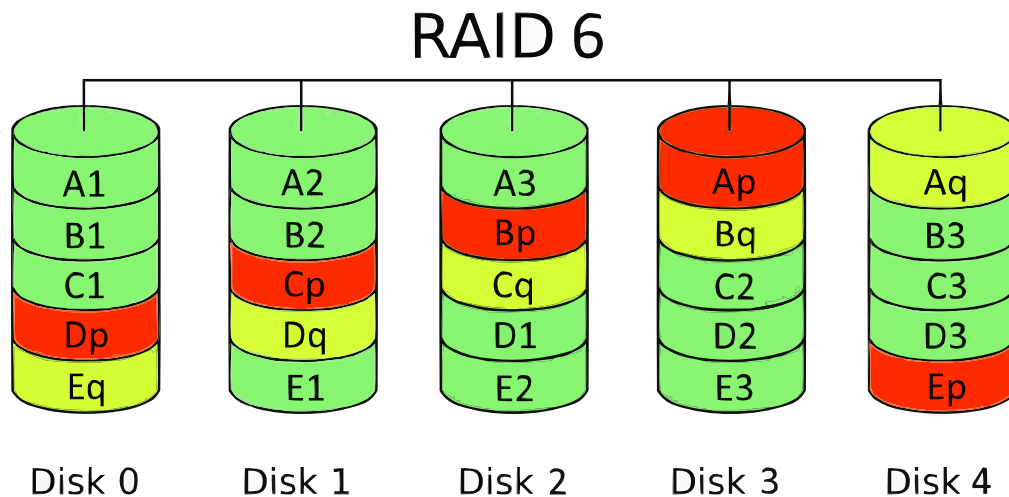
Сама контрольная сумма представляет собой операцию XOR всех остальных стрипов в страйпе. Так как операция XOR полностью коммутативна и самообратима, потеряв один диск информации, мы легко сможем восстановить информацию из утерянного диска, проведя XOR всех уцелевших данных в страйпе с блоком паритета. 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, чтобы вновь иметь отказоустойчивость в один диск.

Таким образом, RAID 5 предоставляет хороший баланс между отказоустойчивостью, производительностью и соотношению полезной информации к избыточной.

RAID 6 расширяет RAID 5 за счет добавления еще одного блока паритета. Для вычисления паритета в RAID 6, помимо операции XOR, используются коды Рида-Соломона. Таким образом, RAID 6 предоставляет отказоустойчивость в два диска, вместо одного.

2.2 Обзор существующих аналогов

Для исследования существующих алгоритмов по объединению запросов записи были проанализированы реализации ИО планировщиков 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 [17].

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 [3]. 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. Это гарантирует, 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 [2].

2.2.2 LINUS ELEVATOR

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 или предшествует ему, тогда выполняется объединение. Если попытка слияния завершается неудачей, выполняется поиск возможной точки вставки в очередь. Если такое место найдено, новый запрос вставляется туда, иначе запрос добавляется в конец очереди.

2.2.3 DEADLINE

Планировщик ввода-вывода DEADLINE работает аналогично 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.

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. Когда приходит запрос на запись, ZFS сама подбирает страйп с идеально подходящей длиной равной длине пришедшего запроса [18].

Таким образом, нет необходимости объединять запрос, если он уже по приходе является объединенным для соответствующего страйпа. Та-

кой подход является намного более эффективным на больших объемах данных по сравнению с read-modify-write в классическом RAID-массиве, однако на маленьких объемах данных тратится большое количество времени на запись метаданных [4].

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.

2.3.1 Основа SPDK

Основой SPDK является асинхронный NVMe драйвер пользовательского пространства, работающий в режиме опроса. Это обеспечивает параллельный, zero-сору 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 [7, 11]. 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 реализуют ту или иную часть 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: `malloc`, RAID и другие, а также написаны заготовки для реализации других блочных устройств [5].

2.3.2 Многопоточность и опросчик

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 система опросов `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 [6].

3 Реализация

Реализация RAID-массивов в SPDK находится в директории `module/bdev/raid`. Именно в ней и будет происходить перехват и объединение запроса.

3.1 Перехват запроса

У всех RAID-массивов в SPDK есть структура модуля `raid_bdev_module`, которая описывает основную информацию о массиве, и какие операции он может выполнять. В ней есть поле с указателем на функцию обработки запроса на запись или чтение. Перехват состоит в том, чтобы подменить эту функцию, но при этом добавив новое поле в структуру, где будет находиться указатель на старую функцию обработки запроса.

Указатель `submit_rw_request` подменяется на нашу функцию, именно туда приходят все запросы на чтение и запись. Указатель `poller_request` был создан нами и содержит стандартную функцию обработки запросов. Необходимо уметь на неё ссылаться, чтобы отправлять туда объединенные запросы.

Стоит отметить, что если массив будет создан без функции объединения запросов или если поступит запрос не на запись, тогда перехват будет чисто формальным и просто вызовет функцию штатной обработки запроса `poller_request`.

3.1.1 Сохранение запросов в буффер

После перехвата, запросы сохраняются в буффер. Он состоит из двух структур данных: хеш-таблица и красно-черное дерево. Деревья хранятся в хеш-таблице и несут информацию про каждый отдельный страйп. Опишем структуру дерева:

Узел `raid_write_request`

- `RW_ENTRY` — define макрос SPDK, который раскрывается в структуре, в которой находятся ссылки на ближайших родственников: папа и два сына

- `addr` — адресс запроса, является ключом для дерева
- `raid_io` — структура SPDK, которая описывает запрос на чтение или запись

Дерево `raid_request_tree`

- `RV_HEAD` — define макрос SPDK, который раскрывается в структуру, в которой находится узел дерева, являющийся корнем дерева
- `size` — текущий размер дерева.
- `last_request_time` — время в микросекундах, когда пришел последний запрос для данного страйпа.
- `is_pollable` — булева переменная, которая показывает можно ли сейчас объединить запрос с помощью опросчика.
- `merge_request_poller` — структура опросчика.
- `raid_bdev` — указатель на структуру, которая описывает RAID массив.
- `stripe_key` — ключ для хеш-таблицы с индексом страйпа.

Таким образом, дерево содержит всю необходимую информацию для объединения запросов. А хеш-таблица делит деревья по страйпам.

3.1.2 Регистрация опросчика

Дерево создается, когда запрос на запись приходит к страйпу впервые. Вместе с тем регистрируется опросчик для этого дерева. Его задача состоит в том, чтобы инициировать преждевременное объединение запросов в случае, если запросы не приходят достаточно долго.

Стоит отметить, что "долго" относительно для каждой машины и системы. Этот вопрос не будет ставиться в дальнейшем в данной работе. А лимитом времени перед преждевременным объединением было выбрано эмпирическим путем 5 микросекунд, так как на моей машине,

эта константа давала наименьшую задержку и при этом не заставляла объединять запросы раньше времени. Нужно иметь ввиду, что эта цифра может существенно разниться от окружения и вычислительной машины.

3.1.3 Особенности перехвата

Не будет лишним сказать, что весь описанный выше процесс работает только, если выбранный механизм ввода-вывода поддерживает глубину очереди. Иначе, будет создаваться только одно дерево и отрабатывать оно будет только по опросчику.

Например, в утилите `fio` по умолчанию стоит синхронный (`ioengine=sync`) механизм ввода-вывода. Он не поддерживает глубину очереди и не пришлет новый запрос, пока не получит ответ о завершении предыдущего.

3.2 Объединение запросов

Функция объединения запросов, может быть вызвана из двух других функций: мгновенно из функции перехвата при наборе `full-stripe`, по таймауту из функции опросчика. Оба этих случая работают одинаково, за исключением причины объединения.

3.2.1 Создание нового запроса и его обработка

Для объединения запросов было принято решение создать новую структуру запроса, в которой будут ссылки на запросы, из которых он состоит. Регистрации нового запроса в `SPDK` не происходило, потому что в таком случае данный запрос необходимо было бы подтвердить и освободить структуру с помощью функций `SPDK`, а так как такого запроса изначально не существовала, неизвестно, к чему это приведет.

Функцией обработчиком данного запроса будет выступать стандартная функция обработки запросов чтения и записи данного `RAID`-массива, указатель, на которую был переопределен в `poller_request`

Новый запрос строится по следующему принципу:

- Создается структура описывающая самый обычный запрос;
- Поля, которые не используются в функции обработчике, не заполняются;
- Поля, которые не содержат информацию о данных, но используются в функции обработчике, заполняются значениями из первого запроса в страйпе;
- Поля, которые содержат информацию о данных, аккуратно берутся из всех остальных запросов для этого страйпа. Важно отметить, что информация не копируется, а идет перестановка указателей;
- Создается новое поле, которое указывает на все запросы, из которых состоит новый запрос.

Таким образом создается новый запрос и он отправляется в функцию, на которую указывает `poller_request`.

3.2.2 Функция завершения (completion)

После того, как запрос был выполнен функцией `poller_request`, он отправляется в `completion` функцию. На данном моменте запрос необходимо снова перехватить.

Перехват необходим для того, чтобы очистить структуру нового запроса, а запросы, из которых он состоит, завершить вручную, чтобы отправить их более нижним уровням.

Созданный запрос ни в коем случае нельзя отправлять в функцию завершения, потому что он не был зарегистрирован нижними уровнями. Таким образом, если пользователь отправил какое-то количество запросов, то информация о их статусе выполнения будет корректно возвращена наверх пользователю в том количестве, в котором было бы без объединения.

4 Тестирование

Тестирование внутри фреймворка SPDK основывается на утилите ввода-вывода `fio` [10]. Стоит также отметить, что проводилось именно интеграционное, а не `unit`-тестирование. Это обусловлено, тем что объединение запросов тесно связано с RAID-массивами и не может быть смоделировано опосредованно. Тестирование написанной функциональности проводилось в RAID 5-массиве.

4.1 Условия тестирования

SPDK поддерживает только операционные системы на базе ядра `linux`, поэтому тестирование было проведено на виртуальной машине на основе дистрибутива `Ubuntu server` версии 24.04.1. Виртуальной машине было выделено 8 ядер процессора и 8 гигабайт оперативной памяти. Процессор: 11th Gen Intel(R) Core(TM) i7-11800H 2.30 GHz

4.2 Параметры тестирования

Для оценки корректности и производительности были зафиксированы следующие параметры:

- В качестве блочного устройства был выбран RAID 5-массив, так как он в полной мере позволяет оценить как корректность так и производительность.
- Массив на основе 8 виртуальных блочных устройств.
- Размер записи 100 мегабайт.
- Механизм ввода вывода `libaio` [22], так как он поддерживает глубину очереди, в отличие от стандартного механизма `fio`.

Между тестами менялась лишь глубина очереди, так как она напрямую влияет на количество отработок опросчика. Значения глубины были выбраны следующие: 1, 4, 7, 16. Такие значения были выбраны

исходя из длины full-stripe- 7 при 8 блочных устройств для RAID 5. Запись проводилась в ram диски с помощью ublk [23].

4.3 Сценарий тестирования

Каждый тестовый случай был запущен 30 раз с включенным объединением и выключенным.

1. Создавался RAID 5-массив на основе 8 виртуальных блочных устройств.
2. С помощью утилиты fio проводилась запись блоков в разные страйпы, для оценки работы в случае нетривиальной нагрузки.
3. С помощью утилиты fio проводилась верификация данных, записанных на RAID 5.
4. С помощью утилиты fio проводилась оценка времени

При таком сценарии тестирования можно оценить корректность работы объединения запросов и её производительность.

4.4 Результаты тестирования

Сразу стоит отметить, что тестирование производительности принято оценить прирост или уменьшение производительности относительно RAID 5 без объединения запросов.

Были получены следующие результаты:

Глубина очереди	Время выполнения записи 100 Мб с объединением, мс	Время выполнения записи 100 Мб без объединения, мс	Прирост времени, %
1	2041 ± 114	1535 ± 99	33
4	639 ± 31	559 ± 34	14
7	449 ± 25	489 ± 19	-8
16	415 ± 19	428 ± 23	-3

iodepth = 1 — все запросы выполняются по таймауту через опросчик. Очевидно, что в данном случае объединение запросов будет не

лучше отсутствия объединения. В среднем увеличение времени выполнения по сравнению с обычной работой RAID 5 в таком случае составило почти 33%.

iodepth = 4 — все запросы выполняются по таймауту через опросчик, однако запрос уже поступает объединенный, хоть и не full-stripe. В таком случае увеличение времени выполнения запросов составило чуть больше 14%. Однако стоит отметить, что и количество запросов на чтение заметно снизилось почти в 2 раза. Это объясняется тем, что RAID 5 использует read-modify-write при обновлении контрольной суммы. В данном случае можно использовать на четыре запроса на чтения меньше для каждого страйпа.

iodepth = 7 — в данном случае почти все запросы были объединены по заполнению дерева. Такой случай не только уменьшил время выполнения запросов записи на 8%, но и уменьшил количество запросов на чтение почти в 5 раз.

iodepth = 16 — при такой глубине очереди время выполнение запросов записи уменьшилось на 3%. Такое снижение результатов по сравнению с глубиной 7 можно объяснить тем, что 16 не делится нацело на full-stripe= 7. Таким образом получается, что в среднем одно из трех деревьев обрабатывает по таймауту, а не по наполнению.

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 полиморфный относительно типа RAID-массива алгоритм для перехвата, отслеживания и объединения запросов записи.
- Была предложена и реализована функциональность преждевременного объединения в случае отсутствия нужного количества запросов в течении долгого времени.
- Написанная функциональность была внедрена в SPDK RAID 5
- Было проведено интеграционное тестирование, проверена целостность файлов, записанных с использованием данного алгоритма и проанализированы изменения в производительности.

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] Completely fair queueing. — URL: https://en.wikipedia.org/wiki/Completely_fair_queueing (дата обращения: 2024-09-22).
- [4] Corporation QueTek Consulting. ZFS and RAID-Z recoverability and performance. — URL: <https://www.quetek.com/zfsandraidz.htm> (дата обращения: 21 декабря 2023 г.).
- [5] Documentation SPDK. — Block Device Layer Programming Guide. — URL: https://spdk.io/doc/bdev_pg.html (дата обращения: 21 декабря 2023 г.).
- [6] Documentation SPDK. — Event Framework. — URL: <https://spdk.io/doc/event.html> (дата обращения: 21 декабря 2023 г.).
- [7] Documentation SPDK. — JSON RPC. — URL: <https://spdk.io/doc/jsonrpc.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] Fio documentation. — 2023. — URL: <https://fio.readthedocs.io/en/latest/index.html> (дата обращения: 2023-12-14).

- [11] Group JSON-RPC Working. — JSON-RPC 2.0 Specification. — URL: <https://www.jsonrpc.org/specification> (дата обращения: 21 декабря 2023 г.).
- [12] Peter M. Chen Edward K. Lee Garth A. Gibson Randy H. Katz David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. — 1994. — URL: <https://dl.acm.org/doi/10.1145/176979.176981> (дата обращения: 2024-02-11).
- [13] RAID setup. — URL: https://raid.wiki.kernel.org/index.php/RAID_setup#RAID-4.2F5.2F6 (дата обращения: 21 декабря 2023 г.).
- [14] SNIA. — URL: <https://www.snia.org> (дата обращения: 21 декабря 2023 г.).
- [15] Syngress. Designing SQL Server 2000 Databases. — URL: <https://www.sciencedirect.com/topics/computer-science/parity-information> (дата обращения: 21 декабря 2023 г.).
- [16] Thorbecke Jan. I/O Optimization. — URL: https://www.cscs.ch/fileadmin/user_upload/contents_publications/tutorials/crayXE6_monte_rosa_workshop/09_CSCS_I0_Optimization.pdf (дата обращения: 2024-05-31).
- [17] 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).
- [18] What is RAIDZ. — URL: <https://www.raidz-calculator.com/what-is-raidz.aspx> (дата обращения: 21 декабря 2023 г.).
- [19] Wikipedia. — RAID. — URL: <https://en.wikipedia.org/wiki/RAID> (дата обращения: 21 декабря 2023 г.).
- [20] Wikipedia. — Read-modify-write. — URL: <https://en.wikipedia.org/wiki/Read-modify-write> (дата обращения: 21 декабря 2023 г.).

- [21] bdevdrv. — 2024. — URL: https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html (дата обращения: 2024-02-11).
- [22] libaio repo. — URL: <https://pagure.io/libaio> (дата обращения: 2024-09-22).
- [23] ublk doc. — URL: <https://docs.kernel.org/block/ublk.html> (дата обращения: 2024-09-22).