Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б15-мм

Реализация лог-структурированного блочного устройства в ядре Linux

Гавриленко Михаил

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

Современные системы хранения данных (СХД) предъявляют высокие требования к производительности операций ввода-вывода на жестких дисках (HDD). В качестве методов оптимизации работы могут предлагаться как аппаратные, так и программные средства. Программной абстракцией для работы с HDD в ядре Linux является блочное устройство. Данная технология предоставляет доступ к обработке данных путем взаимодействия с блоками фиксированного размера, которые являются единицами хранения данных. Виртуальные устройства позволяют реализовать более сложную логику поверх некоторого базового накопителя. Расширенную функциональность блочных устройств можно реализовать как модуль ядра Linux.

Несмотря на успехи в оптимизации различных блочных устройств [4], вопросы об оптимизации производительности HDD, при выполнении большого числа случайных операций остаются широким полем для исследований. Случайная нагрузка на HDD приводит к задержкам при доступе к данным и снижает общую производительность. Предлагается свести случайную запись к последовательной, применяя известный подход лог-структурированности. Основная идея данного подхода заключается в перенаправлении каждой операции записи в последовательный журнал (лог), сохраняя новые данные в конце файла. Это позволяет избежать случайные записи на HDD, как следствие, повысить производительность [7]. Лог-структурированный подход уже давно известен, однако до сих пор нерешёнными остаются вопросы, касающиеся его оптимальной реализации и места интеграции в систему. Для определения наиболее эффективного способа его внедрения необходимы дальнейшие исследования. При реализации лог-структурированного подхода важно учитывать такие требования, как высокая скорость записи и чтения данных, эффективное управление пространством журнала и минимизация времени доступа к данным.

Перенаправление записи и последующее чтение из лог-структурированного блочного устройства подразумевают под собой

существование некоторой таблицы соотношения запросов. Немаловажным является выбор структуры данных, на основе которой реализуется данная таблица. В зависимости от специфики СХД требования к структуре, а как следствие, и её выбор, могут меняться. Однако ключевыми остаются: скорость записи, поиска (в том числе последовательного) и чтения. Одной из структур с подобными свойствами является B+ дерево [1]. Оно представляет собой дерево, в котором все данные хранятся на нижнем уровне. Элементы нижнего уровня соединены между собой, что предоставляет возможность быстро находить последовательно лежащие данные. Такая организация и высокая степень ветвистости дерева позволяет эффективно обрабатывать большие объёмы данных, обеспечивая быстрый поиск, вставку и удаление. Помимо B+ дерева существуют и другие структуры, отвечающие подобным требованиям — список с пропусками, хеш-таблица и красно-черное дерево.

Для повышения производительности HDD при большом числе случайных операций, предлагается использовать лог-структурированный подход, который сводит случайные записи к последовательным, перенаправляя их в лог. Необходимо разработать принцип соотношения блоков и выбрать оптимальные структуры данных для его реализации, уделяя внимание скорости поиска и вставки. В качестве структуры данных может быть использовано B+ дерево, которое эффективно обрабатывает данные благодаря своей организации. Однако, также стоит рассмотреть и другие структуры данных.

1. Постановка задачи

Целью работы является изучение способов реализации лог-структурированной адресации в блочных устройствах. Для достижения цели были поставлены следующие задачи:

1. разработать базовое виртуальное блочное устройства с лог-структурированной адресацией;
2. внедрить принцип лог-структурированности на основе различных структур данных;
3. провести тестирование целостности виртуального блочного устройства.

2. Обзор

2.1. Блочные устройства

Блочные устройства — вид устройств, которые предоставляют программную абстракцию накопителя с доступом к данным в виде блоков определенного размера. Данные на жестких дисках (HDD) организованы в виде секторов — минимальных единиц хранения, обычно объемом 512 байт (B). Основной функцией блочного устройства является эффективное и удобное управление данными. Размер блока данных кратен размеру сектора. Преимущественно блоки бывают размера 8 килобайт (KB), 4 KB и 512 B. В ОС на основе ядра Linux блочные устройства представлены как файлы, имеющие некоторые дополнительные свойства. Примером блочных устройств может послужить жесткий диск и иные накопители.

Блочные устройства также могут быть виртуальными. Программная имитация блочного устройства может быть использована в случаях, где требуется физическое устройство, но при этом оно недоступно. Такими случаями могут быть: тестирование, использование виртуального устройства как промежуточного слоя и упрощение управления данными.

В ядре Linux реализован интерфейс *blkdev.h*, позволяющий создавать и управлять виртуальными блочными устройствами.

2.2. Лог-структурированное хранение данных

Лог-структурированность — технология хранения данных, основанная на принципе журнала. Вместо записи в случайные места — все данные записываются последовательно, что обеспечивает высокую скорость записи на устройство [7]. Кроме того, лог-структурированность используется для реализации механизмов мгновенных снимков и копирования исходных данных в другое место при их перезаписи (Copy-On-Write).

Данная технология не является новой. Первые связанные с лог-

структурированностью работы начали появляться еще в конце 1980-х. Лог-структурированность применяется в файловых системах, драйверах блочных устройств, системах кэширования данных и других областях. Каждое применение по-разному взаимодействует с технологией лог-структурированности, используя различные её преимущества. Тем не менее, не все возможности использования были раскрыты.

2.3. Структуры данных

Структура данных — способ организации данных и их хранения, обладающий определенной эффективностью операций доступа к ним. В данной работе нас интересуют структуры данных с эффективными по скорости операциями поиска и чтения, в том числе последовательного. В ходе исследования были выявлены структуры данных, подходящие под критерии, такие как: B+ дерево, список с пропусками, хеш-таблица и красно-черное дерево.

2.3.1. B+ дерево

B+ дерево — структура данных, сбалансированное дерево поиска с переменным количеством потомков в узле. Все значения находятся на нижнем уровне дерева и являются листьями. В отличие от B-дерева, нижние узлы в B+ дереве являются последовательно связанными. Это обеспечивает доступ к следующему элементу за константное время.

Главное преимущество данной структуры данных — быстрая операция поиска. За счет увеличенного коэффициента ветвления высота дерева снижается, что повышает скорость выполнения операций. Как итог, в B+ дереве операции вставки и удаления имеют оценку временной сложности $O(logn)$.

2.3.2. Список с пропусками

Список с пропусками — структура данных, состоящая из множества слоев — списков. Количество списков вариативное. Нижний слой является обычным списком, состоящим из всех элементов. В конце каждого

списка есть специальный элемент, обозначающий конец списка. Каждый слой выше $(i+1)$ — список, в котором элементы слоя i встречаются с некоторой вероятностью p.

Поиск элемента n начинается с первого элемента в верхнем списке и продолжается горизонтально, пока текущий элемент не станет больше либо равен n. Если текущий элемент равен n, то поиск закончен. В ином случае процесс спускается на слой ниже и продолжает поиск по такому же принципу дальше. Если результат поиска — конечный элемент, то элемент n не находится в списке с пропусками.

Как итог, список с пропусками обеспечивает операции записи, чтения и удаления, в среднем с оценкой временной сложности $O(logn)$ и $O(n)$ в худшем случае.

2.3.3. Хеш-таблица

Хеш-таблица — структура данных, представляющая собой ассоциативный массив, который задает отношение на парах ключ/значение. Для нахождения индекса и последующей вставки значения в определенную ячейку массива используется хеш-функция. Зачастую реализации хэш-таблиц используют неидеальную хеш-функцию. Данная функция обладает недостатком в виде коллизий — отображения разных ключей в один индекс массива с некоторой, не равной нулю, вероятностью.

Существуют две основных реализации хеш-таблиц: с открытой адресацией и на основе цепочек. Ассоциативный массив состоит либо напрямую из пар, либо из других списков с парами. Хэш-таблица обеспечивает довольно высокую скорость операций удаления, поиска и вставки, с оценкой временной сложности $O(1)$ в среднем и $O(n)$ в худшем случае.

2.3.4. Красно-черное дерево

Красно-черное дерево — структура данных на основе бинарного дерева поиска, с некоторыми модификациями. Каждый узел обладает определенным цветом — красным или черным. Дерево обладает такими свойствами, как: корень дерева всегда черный; потомки красного

узла являются черными; у любого узла с как минимум одним пустым потомком — одинаковое количество черных узлов-предков.

В отличие от обычного бинарного дерева поиска, операции усложнены балансировкой, для соблюдения свойств дерева. Благодаря балансировке, красно-черное дерево имеет временную оценку операций вставки, удаления и поиска лучше, чем бинарное дерево поиска. Операции вставки, удаления и поиска обладают оценкой временной сложности $O(logn)$ и в среднем, и в худшем случае, вместо $O(logn)$ и $O(n)$.

2.3.5. Неоднозначность выбора и оценки эффективности структур данных

На первый взгляд, некоторые из структур могут показаться одинаковыми с точки зрения оценки временной сложности, однако существуют различия в пространственной сложности и области применения. Например, B+ дерево эффективно использует память при хранении больших объемов данных, что делает его популярным в системах хранения и базах данных. В то же время, хеш-таблицы обеспечивают быструю работу с ассоциативными массивами, но могут требовать дополнительные ресурсы для управления коллизиями. Списки с пропусками, в свою очередь, имеют оценку временной и пространственной сложности не лучше, чем B+ дерево. Однако их применение довольно популярно при необходимости параллельного взаимодействия с данными [6]. Важно отметить, что при использовании лог-структурированного подхода выбор структуры данных для хранения соотношения адресов может быть неоднозначным и зависеть от специфики реализации и характера данных.

2.4. Обзор существующих решений

Среди открытого ПО существуют реализации по-разному подходящие к лог-структурированности, такие, как Sprite LFS (Sprite Log-structured File System), LD (Logical Disk), DCD (Disk Caching Disk) и ZFS (Zettabyte File System).

2.4.1. Sprite LFS

Одним из первых примеров использования технологии полноценного лог-структурированного хранения является файловая система Sprite LFS. В отличие от предыдущих файловых систем, использовавших лог-структурированность, Sprite LFS пользуется журналом в качестве основного места хранения данных [7]. Это увеличивает производительность записи и упрощает восстановление после сбоев (например, отключения питания или программных ошибок). При сбое несколько последних операций с диском могут оказаться не обработаны. В отличие от традиционных файловых систем семейства ОС Unix, лог-структурированная файловая система содержит информацию о последних операциях в конце лога, что предотвращает долгий анализ метаданных всех операций, результат которых хранится на диске [7].

Минусы Sprite LFS включают частые операции очистки, которые используют значительную часть пропускной способности диска и могут снижать производительность. Кроме того, система неэффективна при работе с маленькими файлами и случайным доступом [7]. Также стоит отметить сложность интеграции, которая обусловлена совершением большого количества изменений в ОС [3].

2.4.2. LD

Logical Disk — одна из первых реализаций лог-структурированного хранения на уровне, ниже файловой системы. LD представляет собой некоторую оптимизацию файловых систем, путем введения нового интерфейса, разделяющего работу файловой системы и управление дисками [2]. Подход был уникален, ввиду легкости разработки, адаптивности под различные требования и эффективного обхода бутылочного горлышка пропускной способности, описанного в исследовании по изучению и применению лог-структурированного подхода [5]. Дополнительно на уровне логического диска, без вмешательства файловой системы, был интегрирован механизм сжатия для оптимизации использования дискового пространства. Это позволяет более эффективно ис-

пользовать дисковое пространство, особенно для хранения редко изменяемых данных. В работе была достигнута основная цель — снижение общего количества операций поиска на HDD.

Для хранения отношения информации о блоках был использован обычный список, который обладает неоптимальной скоростью поиска. Преимущества и недостатки такого подхода не были детально изучены. Как итог, LD остался примером хорошей технологии исключительно для небольших систем.

Тем не менее, работа ввела множество новых понятий, которые впоследствии стали результатами других исследований. Одним из них стала обработка частичных сегментов [3].

2.4.3. DCD

Disk Caching Disk — технология использования некоторого промежуточного Cache-Disk (кеш-диск) на основе журнала для оптимизации записи. Данная технология может быть рассмотрена как некоторая оптимизация LD, в которой добавлен промежуточный слой — Cache-Disk. Маленькие случайные записи сначала сохраняются в небольшом RAM-буфере, затем определенной порцией записываются на кэш-диск. Cache-disk использует лог как расширение буффера RAM для хранения информации об изменениях файлов, после чего, когда система не загружена, контроллер передает данные на основной диск (рис. 1).

Это решение повышает производительность записи по сравнению с традиционными дисковыми системами и не требует изменений в операционной системе. DCD обладает высокой надежностью и способностью сохранять временную локальность I/O операций. Как итог, было получено существенное улучшение производительности при небольших дополнительных затратах.

Однако сфера применения DCD довольно ограничена. Технология эффективно работает с небольшими объемами данных и локальными операциями записи, характерными для небольших систем. Использование небольшого кэша снижает затраты, но ограничивает возможность масштабирования для больших систем. Основной акцент DCD делал-

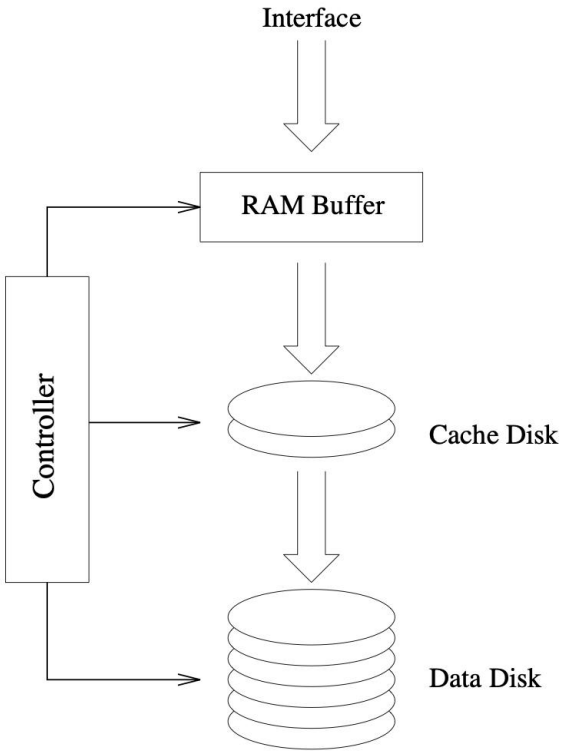

Рис. 1: DCD состоящий из 2 отдельных HDD [3]

ся на снижение задержек, а не на повышение пропускной способности. Кроме того, небольшие системы проще адаптировать под архитектуру DCD с минимальными изменениями.

2.4.4. ZFS

Zettabyte File System — продвинутая файловая система, разработанная с целью предоставления высокого уровня целостности данных, масштабируемости и производительности. ZFS не является традиционной файловой системой. ZFS состоит из двух основных частей — менеджера томов и файловой системы. Вместо фиксированного разделения пространства между томами ZFS объединяет доступные диски в единое хранилище, называемое storage pool (единое общее хранилище). Такой подход позволяет оптимально управлять дисковым пространством. Кроме того, ZFS позволяет задавать уникальные свойства для каждой отдельной файловой системы. ZFS предлагает довольно широкий спектр свойств, например: сжатие, дедупликация, шифрование, мгно-

венные снимки и так далее. Это позволяет удобно создавать множество отдельных файловых систем или наборов данных.

Данная файловая система пользуется принципом лог-структурированности в некоторых частях своей функциональности, например, для реализации Copy-On-Write. Данная механизм предотвращает повреждение исходных данных в случае сбоя. Сегментация помогает записывать данные последовательно, минимизируя фрагментацию, а управление синхронными записями, основанное на этом принципе, обеспечивает быструю фиксацию изменений, сохраняя их целостность [1].

Однако у проекта есть некоторые минусы, например, большая нагрузка на ресурсы, в силу масштабности и сложности реализации. Высокий уровень предоставляемой абстракции усложняет процесс обработки данных и не дает доступа к более низкоуровневому управлению.

[1]The Z File System Docs: https://docs.freebsd.org/en/books/handbook/zfs/

3. Реализация

Далее будет представлена реализация драйвера виртуального блочного устройства с поддержкой лог-структурированной адресации. Процесс разработки можно разделить на три этапа. Для начала необходимо реализовать базовую функциональность лог-структурированного блочного устройства. Также необходимо добавить поддержку обработки запросов разного размера. Третий этап будет нацелен на интеграцию различных структур данных в драйвер блочного устройства.

3.1. Создание базового лог-структурированного виртуального блочного устройства

Предлагается реализовать лог-структурированное хранение на основе взаимодействия двух блочных устройств. Первое блочное устройство — виртуальное, будет принимать I/O (входной/выходной) запрос, создавать его копию и перенаправлять её во второе устройство. Второе блочное устройство — целевой накопитель, который будет принимать и хранить обработанные запросы от первого блочного устройства. Реализация подобного принципа требует создания дополнительного виртуального блочного устройства и структуры данных, в которой будет хранится информация о соотношении адресов секторов начала входных запросов, до и после перенаправления. Интерфейс для удобного взаимодействия с блочными устройствами `blkdev` реализован в ядре Linux.

3.1.1. Инициализация блочных устройств

Для создания виртуального блочного устройства, сначала необходимо зарегистрировать в системе определенное название и уникальный номер устройства, идентифицирующий его, с помощью функции `register_blkdev()`. Помимо регистрации необходимо создать диск, который будет прикреплен к уникальному номеру, используя функцию `alloc_disk()`, которая возвращает структуру `gendisk`. После инициализации `gendisk`, структура будет содержать информацию о доступ-

ных операциях, размере, уникальных идентификаторах и мета-данных блочного устройства.

Так как в драйвере реализуется функциональность добавления нескольких блочных устройств — каждое блочное устройство будет иметь разное название, с уникальным номером в конце, к примеру ”lsvbd1”, ”lsvbd2” и так далее. Для управления множественными виртуальными блочными устройствами и хранения связи виртуального устройства с целевым, в драйвере создается список из вспомогательных структур `bd_manager` (рис. 2).

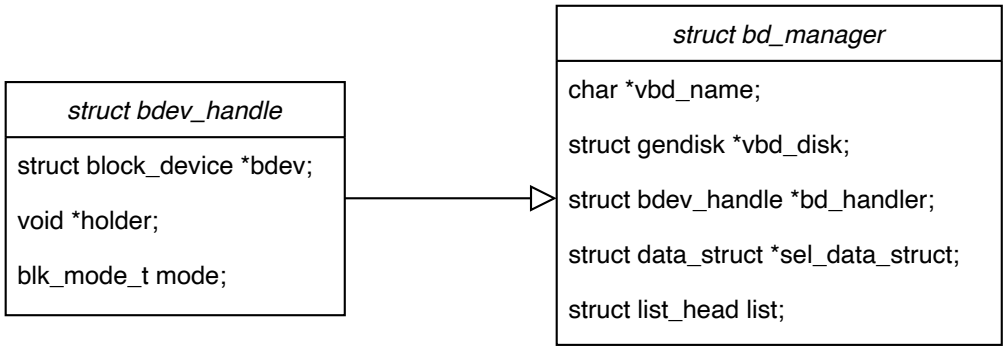

Рис. 2: Структура хранящая описание блочных устройств

3.1.2. Обработка запросов чтения и записи

Ключевым уровнем взаимодействия между блочными устройствами является BIO layer (Block I/O слой). BIO layer является набором функций для взаимодействия со структурой BIO [2]. Данная структура описывает запросы ввода-вывода блочной подсистемы. Один из ключевых элементов BIO layer — структура `gendisk`, представляющая описание блочного устройства. Для обработки запросов используется функция `submit_bio()`, которая принимает на вход структуру BIO и передает соответствующему устройству. Именно в этой функции осуществляется ключевая функциональность лог-структурированного виртуального блочного устройства. Во-первых, для перенаправления запроса необходимо изменить получателя запроса с первого блочного устройства на

[2]A Block layer instroduction part 1: the bio layer: https://lwn.net/Articles/736534/

конечное блочное устройство. Такие действия не рекомендуется совершать вручную, потому что запросы могут принадлежать различным устройствам, которые рассчитывают на сохранность определенных полей в запросе. Для перенаправления рекомендуется использовать функцию `bio_alloc_clone()`, которая создаст новую копию BIO запроса, перенаправляя её в другое указанное блочное устройство. Во-вторых, в связи с внедрением принципа лог-структурированного хранения, запросы чтения и записи требуют различной, отдельной реализации.

Для хранения информации о входящих запросах была введена структура `redir_sector_info`, хранящая информацию о размере и итоговом адресе блока входящего запроса. Хранение пар в формате "ключ-значение", где ключом является начало исходного блока, а значением структура `redir_sector_info`, осуществляется в одной из выбранных структур данных.

Запросы чтения и записи обрабатываются в отдельных функциях `setup_read_from_clone_segments()` и `setup_write_from_clone_segments()`. На базовом уровне данные операции не сильно отличаются друг от друга. Обработка операции записи заключается в нахождении следующего свободного блока, который становится итоговой целью BIO. Из-за особенности технологии лог-структурированного хранения блоки данных лежат последовательно, а значит следующий свободный блок вычисляется как конец предыдущего. Если по адресу блока уже лежат какие-то данные, то они перезаписываются. Иным образом обрабатываются запросы чтения. В начале производится проверка на наличие адреса блока в структуре данных. Если адрес блока не найден, то в структуру данных записывается соотношение адреса блока со следующим свободным блоком. В обратном случае адрес блока операции чтения не изменяется.

3.2. Операции с блоками различного размера

Современные системы хранения данных оперируют данными, организованными в запросы разного размера. В различных сценариях раз-

меры запросов могут варьироваться. К примеру, для записи и хранения большого объема данных — выгодно использовать блоки большого размера. Это поможет снизить затраты на дополнительные действия по обработке запросов. С другой стороны, некоторые операции могут требовать повышенной гранулярности для более точной обработки. Обеспечение поддержки операций разного размера позволит пользователю более гибко взаимодействовать с виртуальным блочным устройством.

Поддержка операций с блоками разного размера основывается на делении и корректном распределении частей блока операции чтения в адресном пространстве жесткого диска. Хранение информации о размере блока позволяет быстрее обрабатывать запросы в том случае, если размер запроса на чтение совпадает с размером записи.

Изменения коснутся обработки операции чтения в `setup_read_from_clone_segments()`. Необходимо рассмотреть случай, когда адрес начала блока не находится в выбранной структуре данных. Так как в структуре данных записываются отношения адресов начала блока, то адрес начала с некоторым смещением, не выходящим за рамки блока, не будет найден. Данный случай должен быть обработан корректно, для чего вводится функция нахождения в структуре данных максимального адреса блока, меньшего исходного адреса блока при чтении. Если конец исходного блока выходит за рамки найденного адреса, то исходный BIO запрос делится на две части. Первая — часть запроса, не выходящая за рамки найденного блока. Вторая часть — остаток, который рекурсивно обрабатывается функцией `setup_read_clone_segments` 3.

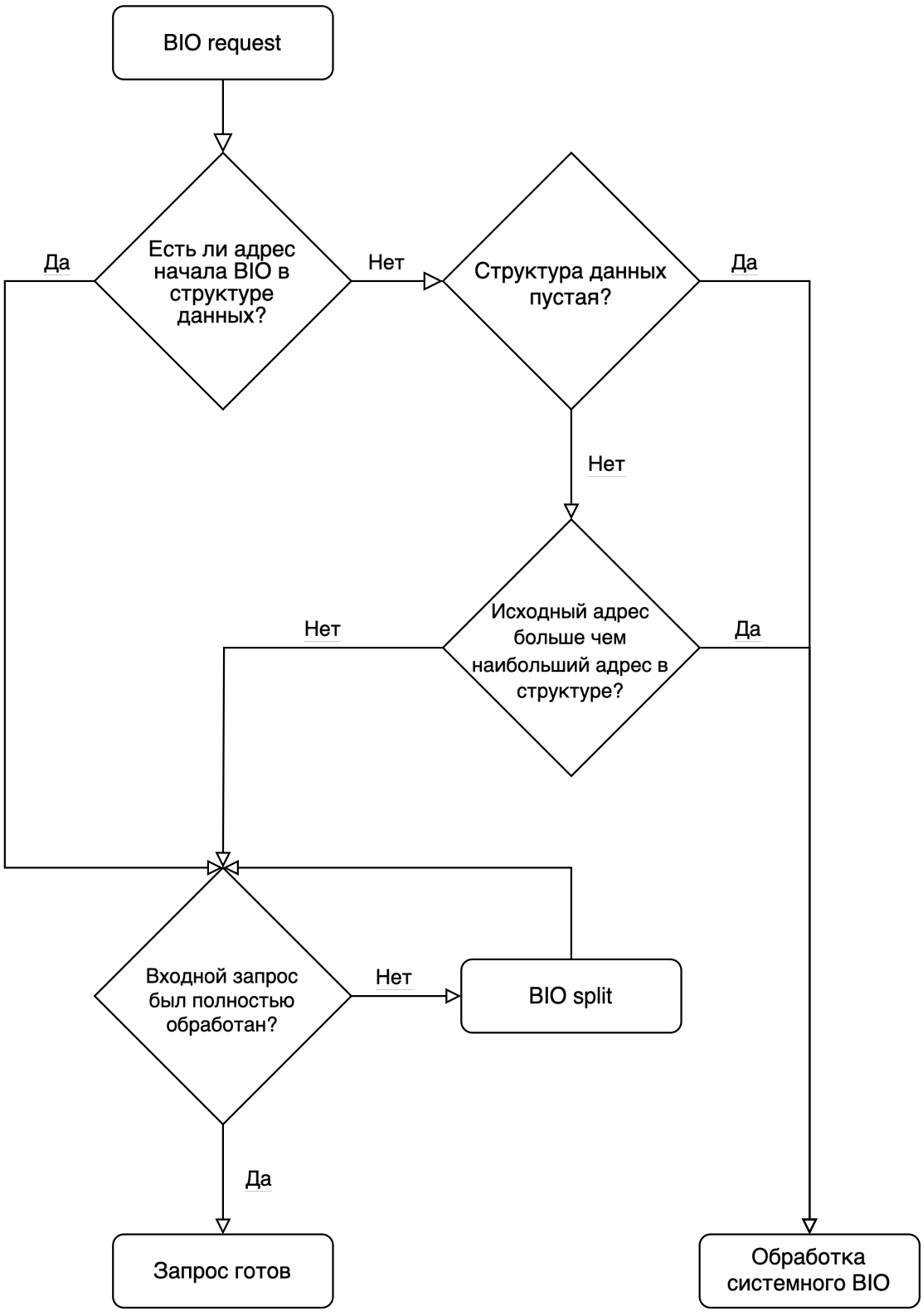

Рис. 3: Процесс обработки запроса на чтение

Также необходимо рассмотреть случай, когда адрес начала блока не находится в структуре данных и не лежит ни в одном из промежутков адресов блоков. Данный случай является системной проверкой и

должен быть обработан корректно.

Случай, когда адрес начала блока находится в выбранной структуре почти идентичен. В отличие от предыдущего, не нужно дополнительно обрабатывать системные BIO запросы.

3.3. Внедрение лог-структурированности на основе различных структур данных

Для тестирования производительности и постановки эксперимента в будущем, предлагается реализовать лог-структурированность на основе различных структур данных. Для этого были выбраны четыре структуры данных, обладающие предположительно эффективными в данном случае операциями чтения, записи и поиска. Далее будут использованы такие структуры, как B+ дерево, красно-черное дерево, список с пропусками и хеш-таблица.

Взаимодействие драйвера блочного устройства со структурой данных реализовано через интерфейс `ds_control`. Данный интерфейс предоставляет функции инициализации, поиска, нахождения последнего и предыдущего элемента, добавления и некоторые другие. Хранение структуры данных осуществляется во вспомогательной структуре `data_structure` (рис. 4). Данная структура является уникальной для каждого виртуального блочного устройства и является частью `bdd_manager`. Это позволяет создавать множественные виртуальные блочные устройства на основе различных структур данных.

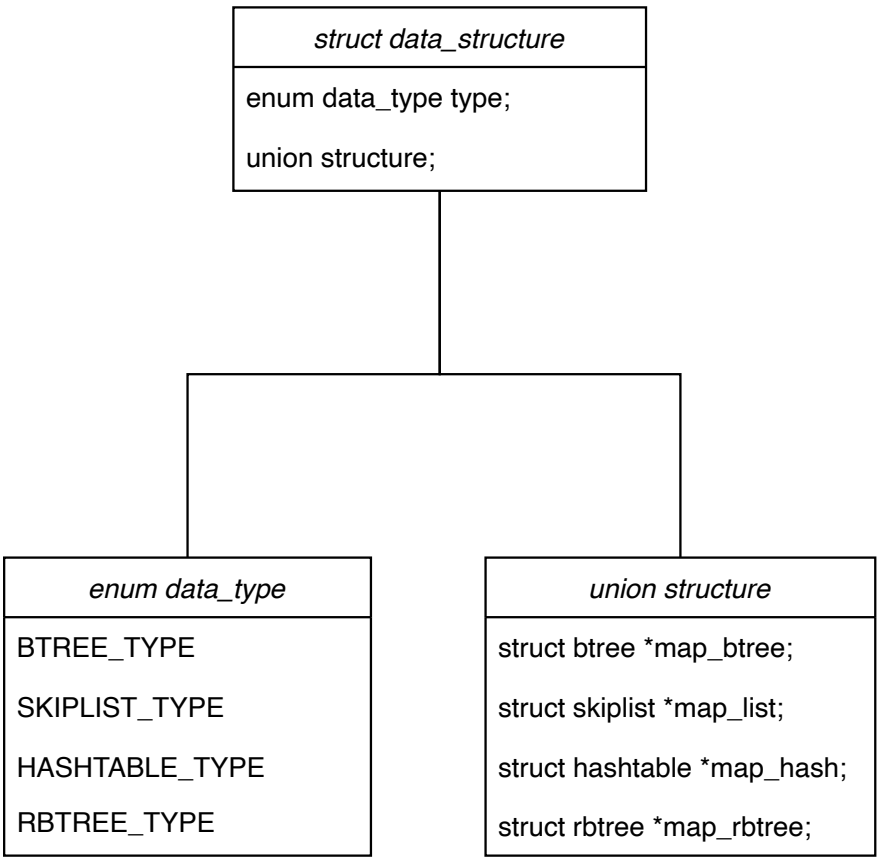

Рис. 4: Структура хранящая информацию о выбранной структуре данных

При внедрении структур данных было принято решение рассмотреть не только существующие в ядре Linux варианты, но и недавние реализации коллег. Таким образом, были использованы готовые реализации хеш-таблицы и B+ дерева. Остальные структуры данных были использованы, опираясь на работы коллег. Тем не менее, все структуры данных были дополнены и переработаны под данный процесс использования. В каждую структуру данных были добавлены функции по нахождению предыдущего и последнего элемента, а также проверки на отсутствие элементов. В силу различия в принципах работы структур данных, подобные методы в будущем могут быть оптимизированы, используя различные преимущества конкретных структур данных.

4. Тестирование целостности

Целостность виртуального блочного устройства является одним из ключевых требований для его применения в системах хранения данных. Данный этап является необходимым, так как в будущем планируется сравнить производительность лог-структурированного подхода на основе различных структур данных. Основная цель тестирования — убедиться, что записанные данные сохраняются в неизменном виде и корректно читаются. Для реализации тестирования были выбраны два инструмента — утилита GNU Project `dd` и FIO (Flexible I/O Tester).

Одним из важных моментов в тестировании виртуального блочного устройства с помощью утилиты `dd` и FIO — использование опции `direct`. Данная опция позволяет обойти page cache (механизм кэширования данных файловой системой в оперативной памяти). Без опции `direct` запросы попадают в кэш и обрабатываются по методике, реализованной конкретной файловой системой. Для автоматизации работы с утилитой `dd` была реализована программа на языке программирования Python. Данная программа предназначена для тестирования всего спектра заданных размеров файлов. Система тестирования также поддерживает проверку всех возможных сочетаний размера блока. Проверка целостности основывается на последовательном сравнении блоков исходного и итогового файлов.

Инструмент FIO является специализированным инструментом для создания потоков ввода и вывода с целью тестирования производительности аппаратных систем для оценки характеристик хранения данных. Помимо проверки целостности, FIO предоставляет довольно широкий спектр возможностей для оценки характеристик блочного устройства [3]. Проверка целостности данных при помощи FIO возможна за счет генерации и добавления контрольной суммы, которая вычисляется по уникальному содержанию блока данных. Однако такой метод не совместим с тестированием целостности при использовании различных размерах блоков операций чтения и записи. Генерация контрольной суммы

[3] fio - Flexible I/O tester: https://fio.readthedocs.io/en/latest/fio_doc.html

производится для конкретного блока и вставляется в его начало. Если запросы чтения и записи имеют разный размер, то при обработке блоков найденная контрольная сумма не будет совпадать. Для проверки целостности данных было принято решение использовать специальный метод проверки по определенному шаблону. Данный метод заключается в записи определенного шаблона внутрь блока. При проверке блоков чтения, размером больше записанных, FIO учитывает размеры и проверяет блоки исходного размера.

Заключение

В рамках данной работы были получены следующие результаты:

- разработано базовое виртуальное блочное устройство с лог-структурированной адресацией;
- внедрен принцип лог-структурированности на основе различных структур данных и добавлен единный интерфейс взаимодействия со структурами данных;
- проведено тестирование целостности реализованного виртуального блочного устройства.

С реализованной функциональностью можно ознакомиться на GitHub [4].

В весеннем семестре планируется протестировать реализованное блочное устройство на сервере и провести анализ реализаций на основе различных структур данных.

[4] LS-BDD: https://github.com/qrutyy/ls-bdd

Список литературы

[1] Curless Brian. CSE 326: Data Structures B-Trees and B+ Trees. — URL: https://courses.cs.washington.edu/courses/cse326/08sp/lectures/11-b-trees.pdf (дата обращения: 1 марта 2008 г.).

[2] De Jonge Wiebren, Kaashoek M Frans, Hsieh Wilson C. The logical disk: A new approach to improving file systems // Proceedings of the fourteenth ACM symposium on Operating systems principles. — 1993. — P. 15–28.

[3] Hu Yiming, Yang Qing. DCD—disk caching disk: A new approach for boosting I/O performance // ACM SIGARCH Computer Architecture News. — 1996. — Vol. 24, no. 2. — P. 169–178.

[4] Mohamed Salem El Sayed. Analysis and Optimization of Storage IO in Distributed and Massive Parallel High Performance Systems. — 2011. — URL: https://www2.informatik.uni-stuttgart.de/bibliothek/ftp/medoc.ustuttgart_fi/MSTR-3196/MSTR-3196.pdf (дата обращения: 15 ноября 2011 г.).

[5] Ousterhout John, Douglis Fred. Beating the I/O bottleneck: A case for log-structured file systems // ACM SIGOPS Operating Systems Review. — 1989. — Vol. 23, no. 1. — P. 11–28.

[6] Parallelizing skip lists for in-memory multi-core database systems / Zhongle Xie, Qingchao Cai, HV Jagadish et al. // 2017 IEEE 33rd International Conference on Data Engineering (ICDE) / IEEE. — 2017. — P. 119–122.

[7] Rosenblum Mendel, Ousterhout John K. The design and implementation of a log-structured file system // ACM Transactions on Computer Systems (TOCS). — 1992. — Vol. 10, no. 1. — P. 26–52.