Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Шалымов Роман Сергеевич

Разработка файловой системы с поддержкой дедупликации данных

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Рецензент:
ст. преп. Ярыгина А. С.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering Chair

Roman Shalymov

Development of a file system with support of data deduplication

Bachelor's Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
professor Andrey Terekhov

Reviewer:
assistant Anna Yarygina

Saint-Petersburg
2016

Оглавление

Введение **4**

Постановка задачи **5**

1. Обзор существующих решений **6**
1.1. LessFS 6
1.2. SDFS 6
1.3. BTRFS 6
1.4. ZFS 7

2. Архитектура VFS **8**
2.1. Обзор VFS 8
2.2. Основные определения 9
2.3. Структуры данных VFS 11

3. Проектирование файловой системы **13**
3.1. Структура корневой директории 17
3.2. Управление свободной памятью 17
3.3. Создание файла 18
3.4. Чтение файла 18
3.5. Запись в файл 19
3.6. Удаление файла 21

4. Апробация **22**
4.1. mkfs 22
4.2. proc директория 22
4.3. Последовательная запись 23
4.4. Произвольная запись 24
4.5. Тест консистентности 24
4.6. Тестирование QEMU образов 25

5. Результаты **28**

Список литературы **29**

Введение

В последнее время тема дедупликации данных является одним из центров внимания производителей и IT администраторов оборудования хранения информации. Она стала особенно актуальной в связи с текущей тенденцией экспоненциального роста данных, которую испытывают большинство современных дата-центров мира.

Дедупликация данных – это процесс нахождения и исключения избыточности данных посредством удаления дублирующихся копий информации на дисковом хранилище. Цель дедупликации заключается в том, чтобы разместить большее количество данных на меньшем пространстве. Одним из примеров применения дедупликации данных может служить замена повторных копий файлов ссылками на единственный экземпляр. Такой тип дедупликации называется файловым.

Традиционно выделяют несколько типов дедупликации в зависимости от уровня реализации – на уровне файлов, блоков данных и даже на битовом уровне. Механизм дедупликации на файловом уровне уже давно реализован и приобрел широкое применение в традиционных UNIX-подобных файловых системах, таких как ext2, при помощи отдельного хранения мета-информации о данных (концепция inode) и самих данных (концепция data block)[2, 3, 4, 5]. Это позволяет иметь одну копию файла на дисковом пространстве и хранить множество линков на единственный образ. Однако этот метод не применим даже при незначительном изменении файла. Одним из примеров служит хранение образов виртуальных машин множества пользователей. Изначальный образ является общим для всех пользователей, но изменения, вносимые каждым пользователем, являются индивидуальными, что не позволяет применять метод файловой дедупликации в данном случае, поскольку на уровне стандартной файловой системы UNIX могут адресоваться только линки на файлы в целом, а не на измененения в них. Поэтому в этом случае могла бы быть применена дедупликация на блочном уровне.

Также методы дедупликации делятся на inline и offline типы. Дедупликация, осуществляющаяся в момент записи данных на диск, называется inline дедупликацией, а та, которая реализуется отдельным процессом в последующее время, называется offline дедупликацией.

Постановка задачи

Целью данной работы является проектирование и реализация файловой системы на уровне модуля ядра с поддержкой inline дедупликации данных для семейства операционных систем с ядром Linux. Для достижения поставленной цели были выделены следующие подзадачи.

1. Изучение интерфейсов VFS подсистемы ядра Linux
2. Разработка архитектуры файловой системы: проектирование дисковой разметки и используемых структур файловой системы для дедупликации данных, а также операций над ними
3. Реализация userspace утилиты mkfs для создания дисковой разметки спроектированной файловой системы на блочном устройстве
4. Реализация драйвера файловой системы с поддержкой дедупликации данных для ядра Linux
5. Апробация реализованной функциональности:
    - Реализация драйвера, обеспечивающего доступ к внутренней статистике использования смонтированной файловой системы с помощью механизма /proc директорий;
    - Сравнительный анализ различных подходов к организации процесса дедупликации данных для файловых систем.

1. Обзор существующих решений

На данный момент существует несколько файловых систем с поддержкой дедупликации данных.

1.1. LessFS

LessFS – это Fuse-Based файловая система, то есть файловая система пространства пользователя (user-space). lessfs перед записью блока данных может определить, являются ли данные излишними путем расчета уникального значения 192-битовой tiger хэш-функции от каждого блока данных, записанных на диск. Когда lessfs определяет, что блок является уникальным и его нужно записать на диск, файловая система сначала сжимает блок при помощи LZO или QUICKLZ алгоритмов сжатия, а затем размещает его на дисковом хранилище.

1.2. SDFS

sdfs – распределенная user-space файловая система, разрабатывающаяся в рамках открытого проекта opendedup. Она представляет собой высокоуровневую прослойку между файловой системой и пользователем, которая написана на языке Java, и не является файловой системой в прямом смысле этого слова. В связи с этим, одним из минусов можно отметить высокое потребление ОЗУ – до 1 гигабайта на терабайт данных на диске. Также стоит отметить, что она реализована с использованием механизма FUSE для Unix-подобных операционных систем, который представляет собой подключаемый модуль ядра, транслирующий вызовы функций из библиотеки libfuse в соответствующие системные вызовы подсистемы VFS ядра операционной системы.

1.3. BTRFS

btrfs поддерживает offline дедупликацию посредством утилит bedup и duperemove. Дедупликация реализована с использованием особенности btrfs, которая позволяет клонировать данные одного файла в другой [6]. Клонированные промежутки становятся общими на диске, что позволяет сохранить дисковое пространство. Этот подход имеет несколько минусов: перед клонированием требуется блокировка файлов (file locking) в userspace пространстве, чтобы их содержимое не могло измениться между временем сравнения данных и временем их клонирования. Это реализовано выставлением immutable аттрибута на файл, сканированием /proc директории на наличие процессов, которые имеют доступ на запись в файл (просмотром файловых дескрипторов или отображения памяти процесса). Если таковые процессы отсутствуют, то только в этом случае производится сравнение и клонирование данных.

1.4. ZFS

zfs – это локальная файловая система, разработанная компанией Sun Microsystems Inc. первоначально для операционной системы Solaris. Она использует модель storage пулов для управления физической памятью, что значительно уменьшает связанные с ней проблемы администрирования разделов, резервирования и добавления новых дисковых носителей [1]. ZFS предоставляет дедупликацию на блочном уровне, что гармонично связывается с 256-битными блочными контрольными суммами, которые обеспечивают уникальные сигнатуры для всех блоков в storage пуле при условии, что хэш-функция криптографически сильна. По умолчанию в качестве криптографической функции используется SHA256 алгоритм.

2. Архитектура VFS

2.1. Обзор VFS

Virtual Filesystem (VFS) – это подсистема ядра, реализующая интерфейсы для работы с файлами, предоставляемые user-space программам. Все файловые системы опираются в своей работе на VFS, позволяя файловым системам не только существовать, но и взаимодействовать между собой. Данная подсистема позволяет программам использовать стандартные системные вызовы Unix для чтения и записи данных на различные устройства, обрабатываемые различными драйверами файловых систем.

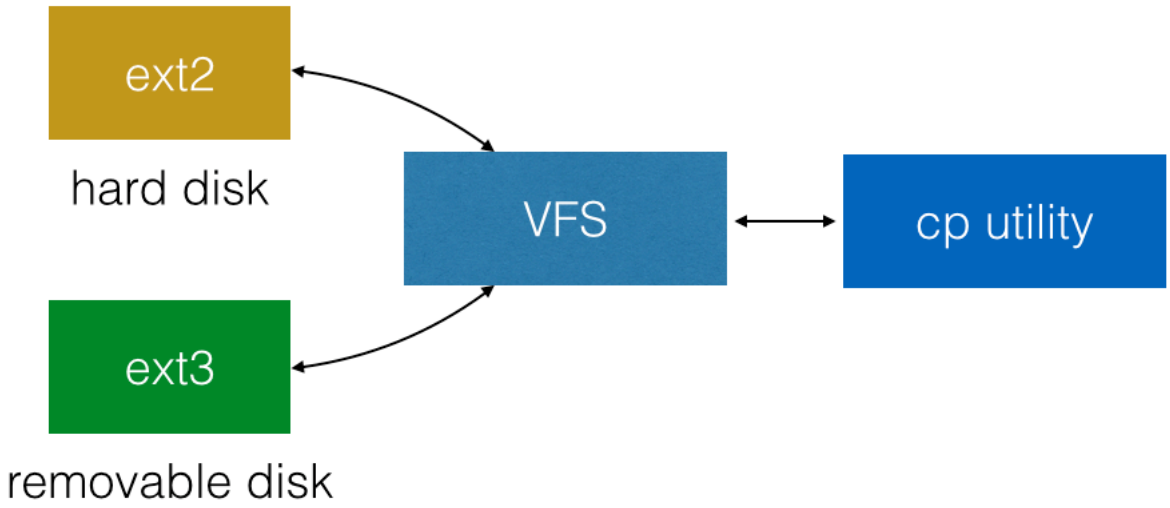

Рис. 1: Механизм копирования данных между устройствами

На рис. 1 схематически представлен общий механизм копирования данных с жесткого диска, монтированного как ext3 на внешний диск, монтированный как ext2, при помощи user-space программы, в данном случае утилиты cp.

Таким образом, новые файловые системы и устройства хранения данных могут быть встроены в Linux, не требуя переписывания или даже перекомпилирования существующих программ.

Уровень VFS предоставляет универсальную, общую для все файловых систем модель файла и методы работы с ней. Это обеспечивается определением основных концептуальных интерфейсов и структур данных, которые все файловые системы обязаны поддерживать. Непосредственно сам код файловой системы скрывает все детали реализации предоставленных интерфейсов. Для подсистемы VFS, а также для других компонент ядра все файловые системы выглядят идентичными. Как результат, ядро может универсально работать с любой файловой системой.

Для описания механизма обмена данными между пользовательскими процессами и устройством хранения требуется определить логические уровни, на которых про-

исходит взаимодействие. Сначала user-space процесс инициирует системный вызов, связанный с работой с файлом, который обрабатывается универсальным обработчиком VFS этого системного вызова. Данный обработчик при помощи поиска соответствующей точки монтирования опередеяет, на какой файловой системе расположен файловый дискриптор файла, и затем вызывает конкретный метод драйвера определенной файловой системы.

На уровне драйвера файловой системы происходит непосредственное взаимодействие с устройством хранения информации. Данный уровень описывает структуры хранения данных и методы работы с ними.

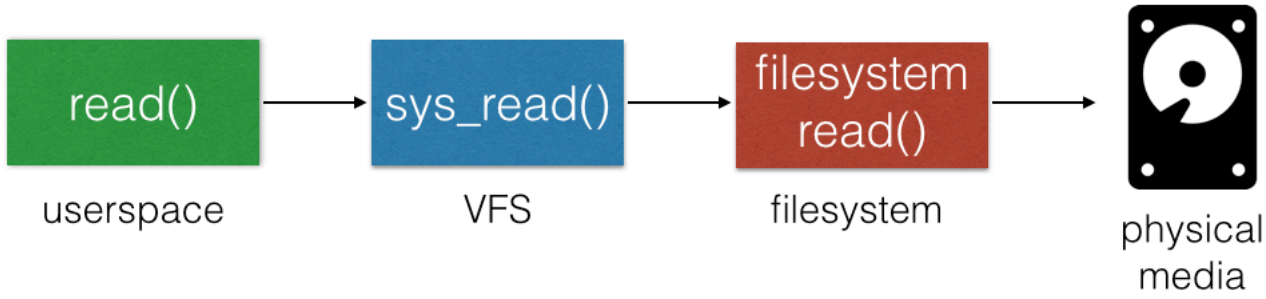

Рис. 2: Цепочка вызовов read()

На Рис. 2 изображена цепочка вызовов на примере системного вызова read(): сначала происходит обработка системным вызовом VFS sys_read(), который вызывает обработчик чтения файловой системы, который, в свою очередь, уже считывает данные с физического устройства.

2.2. Основные определения

Предлагается рассмотреть основные абстракции, исторически предоставляемые Unix, связанные с работой файловых систем:

- файлы
- записи каталога (directory entries)
- айноды(inodes)
- точки монтирования (mount points).

Файловые системы содержат файлы, директории и связанную с ними контрольную информацию. Типичные операции, производимые над файловыми системами, – это создание, удаление и монтирование. В Unix файловые системы монтируются в

специальную точку монтирования (mount point) в глобальной иерархии, также называемой пространством имен файлов. Это позволяет всем монтированным файловым системам существовать как сущности в едином дереве. В отличие от концепции, используемой в DOS и Windows, которая разбивает пространство имен файлов на имена устройств, присваиваемых каждому носителю с файловой системой, как, например, C :.

Файл – упорядоченный набор байт. Первый байт ассоциируется с началом файла, а последний – с концом файла (EOF). Типичные операции, производимые над файлами, – это создание, удаление, чтение и запись.

Файлы организуются в директории. Директории могут также содержать другие директории, называемые поддиректориями. То есть директории могут быть вложенными, таким образом формируя путь. Каждая компонента пути называется записью каталога (directory entry). В Unix директории являются файлами, которые отличаются от обычных файлов всего лишь одним флагом в структуре inode, представляющей собой всю мета-информацию о файле. Файл, соответствующий директории, содержит в себе список файлов, которые содержатся в данной директории. Таким образом, операции, производимые над обычными файлами, могут быть также произведены и над директориями.

Интерфейс Unix разделяет концепцию файла от ассоциированной с ним информации о самом файле, такой как размер файла, uid и gid владельца файла, дата создания, доступа и модификации. Эта информация часто называется метаданными файла(file metadata).

Вся контрольная информация, связанная с функционированием файловой системы в целом хранится в структуре данных, называемой суперблоком (superblock). Она также часто упоминается как мета-данные файловой системы.

Обычно файловые системы Unix используют эти понятия при реализации физической разметки диска. Например, контрольная информация о всей файловой системе хранится в отдельном блоке на диске, именнуемом суперблоком, мета-данные файла хранятся как айноды в отдельном блоке, именуемом хранилищем айнодов и так далее.

Абстрактный уровень Linux VFS разработан таким образом, чтобы работать с файловыми системами, которые понимают и реализуют описанные концепции. Например, если архитектура файловой системы трактует директории и файлы как различные объекты со своим отличным друг от друга поведением, для уровня VFS они должны быть в любом случае оба представлены как файлы. Или если файловая система физически хранит мета-данные файла вместе с данными файла в одном месте, или же если она вообще не поддерживает концепцию айнода, то драйверу в любом случае необходимо создавать в памяти отдельную структуру inode, ассоциированную с конкретным файлом.

2.3. Структуры данных VFS

Существует четыре основных структуры данных (объекта), предоставляемых уровнем VFS

- структура superblock, которая хранит информацию о монтированной файловой системе
- структура inode, представляющая собой конкретный файл на диске
- структура file, представляющая собой открытый файл в памяти, связанный с конкретным процессом
- структура dentry, представляющая собой запись каталога, то есть одну компоненту пути

Для каждого из типов объектов определена своя структура operations. Данные структуры описывают методы, которые ядро может вызывать над соответствующими объектами:

- структура super_operations, содержащая методы для действий над конкретной файловой системой
- структура inode_operations, содержащая методы для работы с конкретным файлом на диске, такие как create() и lookup()
- структура dentry_operations, содержащая методы для действий над конкретной компонентой пути, как d_compare() и d_delete()
- структура file_operations, содержащая методы, которые процесс может вызывать над открытым файлом, такие как read() и write()

Эти объекты операций реализованы как структуры, содержащие указатели на функции-обработчики, которые оперируют над родительским объектом.

Каждая зарегистрированная в ядре файловая система представлена структурой file_system_type. Данная структура описывает основные характеристики драйвера файловой системы, такие как имя, владелец модуля, функция-обработчик, который будет вызван при монтировании файловой системы, флаги монтирования и так далее. Кроме того, каждая точка монтирования представлена структурой vfsmount. Эта структура содержит такую информацию как ее местоположение в глобальном пространстве имен файлов и флаги монтирования.

Также две структуры fs_struct и file, ассоциированные с конкретным пользовательским процессом, описывают характеристики конкретной файловой системы и файла, открытого данным процессом.

Более детальное описание методов и полей структур можно прочитать в соответствующих книгах [4],[5]

3. Проектирование файловой системы

Проектирование архитектуры файловой системы предполагает разработку используемых структур данных, организации дисковой разметки файловой системы, а также создание алгоритмов для работы с данными структурами при выполнении операций над файлами, такими как чтение и запись.

Прежде всего стоит отметить ключевые особенности разрабатываемой модели файловой системы (далее dedupfs), которые повлияли на дальнейшие архитектурные решения

- файлы разбиваются на блоки (кластеры)
- два одинаковых блока не хранятся
- механизм copy-on-write
- специализация для хранения больших файлов (образы VM)

Поскольку dedupfs имеет специальное назначение для хранения больших файлов, а также с целью упрощения дальнейшей программной реализации было решено ограничить максимальное количество создаваемых файлов числом 128.

Также было решено отказаться от вложенных директорий, поскольку, как было сказано ранее, директории в Unix являются файлами, и возможность создания новых директорий никак не помогает продемонстрировать ключевые концепции разрабатываемой файловой системы и только лишь усложняет программную реализацию. Таким образом общий дизайн dedupfs представляет собой корневую директорию, в которой может быть создано до 128 обычных файлов включительно.

Файловой системе необходимо хранить мета-данные о каждом файле, которые включают в себя информацию о том, какие блоки данных (кластеры) составляют файл, размер файла, его владелец, права доступа, время доступа и модификации. Для хранения такого рода информации файловые системы обычно используют структуру, называемую inode. Dedupfs также использует специальную структуру, далее именуемую dfs_inode с целью избежания конфликта с названием структуры inode интерфейса VFS.

Физическое пространство диска разбивается на блоки фиксированного размера. В первом блоке диска было решено хранить всю ключевую иформацию о файловой системе, а именно размер используемого блока, количество используемых айнодов, магическое число, а также версию файловой системы. В связи с этим данный блок в дальнейшем будет упоминаться как superblock.

Для хранения айнодов файлов было решено использовать второй блок дискового устройства, который содержит в себе массив структур dfs_inode. Далее данный раздел диска будет именоваться таблицей айнодов или inode table.

Здесь необходимо отметить, что размер айнодов обычно невелик, в нашем случае он получился 28 байт. Предполагая 28 байт на один айнод, блок размера 4КБ может содержать в себе до 146 айнодов, что вполне удовлетворяет условиям, налагаемым на разрабатываемую систему.

Одними из ключевых компонент любой файловой системы являются аллокаторы свободных блоков диска и айнодов файлов. В данный момент стоит сделать различие между блоками данных и блоками диска. Когда упоминаются блоки данных, то имеются в виду блоки, хранящие непосредственно данные файлов. Когда упоминается блок диска, то имеется в виду физический блок независимо от целей его использования, то есть в независимости используется он для хранения суперблока, айнодов или же данных файлов.

Существует множество возможных способов аллокации блоков. Например, можно использовать список свободных блоков, голова которого указывает на первый свободный блок, тот в свою очередь указывает на следующий свободный блок и так далее, обозначим данный метод как free list allocation. Вместо него была выбрана популярная в файловых системах структура, называемая битовой картой(bitmap), и использована для реализации аллокатора блоков данных. Битовая карта – это простая структура, каждый бит которой указывает, свободен соответствующий объект/блок (0) или используется (1).

Одной из самых важных задач в разработке структуры dfs_inode было определить, каким образом будет осуществляться доступ к блокам данных файла.

Простейшим возможным решением было бы хранить несколько прямых указателей (адресов диска) внутри айнода. Каждый указатель ссылается на один блок данных, принадлежащих файлу. Однако данный подход весьма ограничен: отсутствует поддержка больших файлов, то есть файлов, размер которых превышает размер блока умноженный на количество прямых указателей. По понятным причинам, этот метод не применяется ни в одной существующей файловой системе.

Для поддержки больших файлов одной из идей является хранить в айноде файла специальный указатель, называемый непрямым указателем (indirect pointer). Вместо ссылки на блок данных, он указывает на блок, содержащий массив указателей, которые в свою очередь ссылаются на блоки данных. В дальнейшем блоки такого типа будем называть indirect блоками. Этот метод организации данных называется многоуровневым индексом.

Предполагая, что размер блока диска равен 4КБ, и блоки диска адресуется 4-байтными адресами, каждый indirect блок может содержать 1024 указателей, позволяя адресовать файлы до 4МБ.

Становится понятно, что чтобы организовать поддержку файлов еще большего размера при этом же подходе, можно добавить в структуру inode еще один блок, содержащий указатели на indirect блоки. В дальнейшем болки такого типа будем

называть double indirect блоками, что позволяет адресовать файлы размером до 4GB.

Аналогичным образом вводится понятие triple indirect блока, который содержит указатели на double indirect блоки, что позволяет потенциально адресовывать файлы размером до 4ТБ.

Многие популярные файловые системы, используют многоуровневый индекс, включая ext2 и ext3, а также UFS или FFS.

В dedupfs было решено использовать именно такой подход к организации хранения данных файла. Каждый айнод файла имеет указатель на triple indirect блок, по умолчанию предполагается, что размер блока имеет стандартный размер 4КБ и ограничение размера файла до 4ТБ является вполне допустимым ограничением. Все эти ограничения являются условными и зависят в первую очередь от размера блока диска. Естественно, если выбрать размер блока равным 8КБ, то максимальный размер файла будет равным 64ТБ.

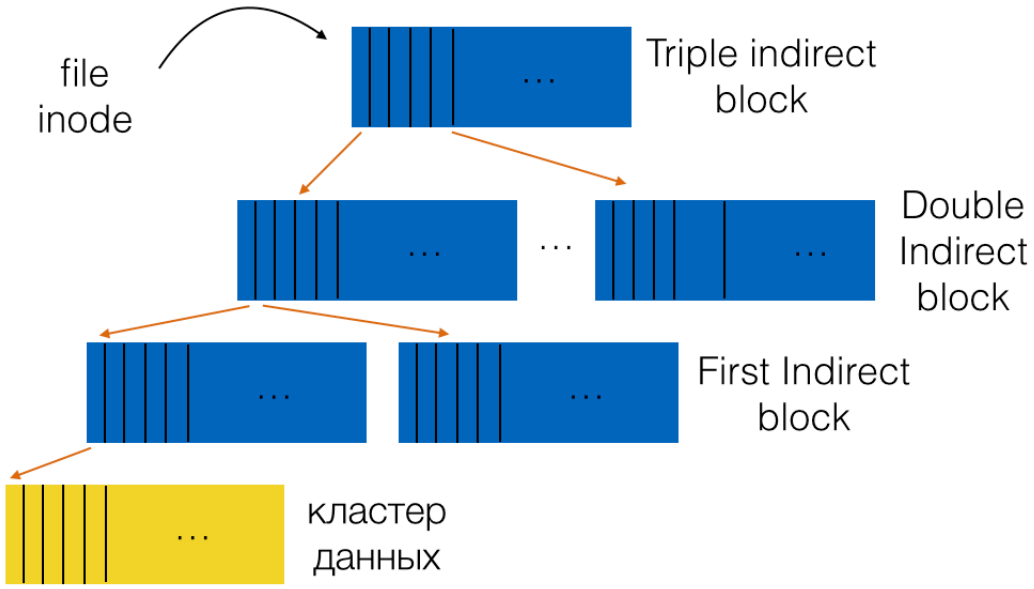

Рис. 3: Структура файла

На рис. 3 изображена внутренняя структура хранения файла в dedupfs.

Другим подходом является использование экстентов вместо обычных указателей на блоки данных. Экстент – это пара указателя на блок диска и длины экстента в блоках. Таким образом, вместо хранения указателя на каждый блок данных, требуется хранить всего лишь один указатель на первый блок данных и соответствующую длину экстента, что существенно сокращает количество хранимой метаинформации файла. Обычно extent-based файловые системы позволяют хранить несколько экстентов для одного файла, поскольку аллокатор памяти может не найти непрерывный участок блоков диска требуемого размера при записи в файл. Примером файловой системы, использующей данный подход, является BTRFS.

Поскольку разрабатываемая нами файловая система должна поддерживать дедупликацию на блочном уровне, то этот метод не может быть применен напрямую.

Также подход, основанный на многоуровневых индексах, является более гибким по сравнению с подходом, основанном на экстентах, однако требует большего количества метаданных, хранимых для каждого файла. Главным недостатком экстентов является повышенная сложность реализации файловой системы.

Существует еще один простой альтернативный метод, состоящий в использовании связного списка. Внутри айнода вместо множества указателей на неявные блоки хранится всего один указатель на первый блок данных файла. Для поддержки файлов большего размера к концу первого блока добавляется указатель на следующий блок данных и так далее.

Как несложно заметить, аллокация блоков при помощи связного списка плохо (линейно) справляется со своей задачей с временной точки зрения при некоторых операциях над файлом, например, при чтении последнего блока файла, или же при произвольном доступе к файлу с помощью системного вызова lseek(). Поэтому некоторые системы для ускорения работы данного метода аллокации хранят таблицу всех ссылок в отдельном месте на дисковом носителе, вместо того, чтобы хранить указатели на следующие блоки данных внутри них же самих. Это позволяет файловой системе загрузить всю таблицу в память и производить кэшированный поиск нужного блока вместо последовательного чтения с диска.

Данная таблица индексируется адресом блока данных, обозначим его за D; ячейка таблицы, соответствующая блоку данных D, содержит следующий указатель, то есть адрес блока данных, следующего за D. NULL-значение в текущей ячейке служит индикатором конца файла (EOF), также могут использоваться и другие маркеры, обозначающие, что соответствующий блок свободен. Наличие такой таблицы указателей при использовании списочной модели аллокации делает более эффективным произвольный доступ к файлу, сначала сканируя в памяти таблицу для нахождения запрошенного блока, а затем производя прямой доступ к диску.

Описанная выше схема является структурой, именнуемой file allocation table или FAT(ссылка на статью). Все семейства FAT файловых систем используют данную схему аллокации. Их отличие состоит только в размерах используемых блоков(кластеров). Стоит отметить, что в архитектуре этой файловой системы отсутсвует понятие айнодов, но есть понятие directory entries, которые хранят метаинформацию о каждом файле и ссылаются напрямую на первый блок данных соответствующего им файла, что делает невозможным создание жестких ссылок (hard links) в Unix-подобных операционных системах.

На рис. 4 изображена схема дисковой разметки dedupfs.

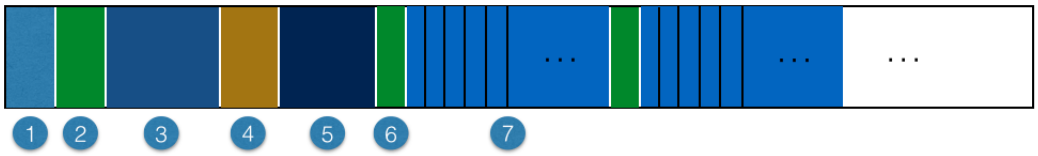

Рис. 4: Дисковая разметка dedupfs: 1 – Суперблок; 2 – Хранилище айнодов; 3 – Битовая карта свободных блоков данных; 4 – Блоки хэш-таблицы, элементы которой содержат номера первых блоков соответствующих цепочек коллизий; 5 – хранилище служебных блоков; 6 – блок, содержащий кластеры для последующих за ним блоков данных; 7 – блоки данных

3.1. Структура корневой директории

В нашей файловой системе, как и в многих других, директория имеет простую организацию; директория содержит список пар (entry name, inode number), где entry name – имя файла, а inode number – идентификатор айнода файла в хранилище айнодов. Для простоты организации директории было решено ограничить длину имени файла 256 символами, как было сделано, например, в NTFS.

Сама корневая директория, как было сказано ранее, является специальным файлом, поэтому имеет свой айнод и соответствующую запись в таблице айнодов.

Поскольку в нашей файловой системе только одна директория, то было решено зарезервировать под нее третий блок диска и аллоцировать дополнительные блоки в общем пространстве блоков данных только при необходимости, то есть при превышении 15 файлов.

Однако существуют и другие способы хранения такого рода метаинформации. Например, XFS хранит директории в форме B-дерева, позволяя операции создания файла, которая должна проверить, что имя создаваемого файла не было использовано до этого, работать быстрее, чем на системах с простым списком.

3.2. Управление свободной памятью

Файловая система должна отслеживать, какие айноды и блоки данных свободны, а какие – нет, для нахождения и аллоцирования свободного места при создании новых файлов и записи в них. Таким образом, управление свободной памятью является важной компонентой любой файловой системы.

В dedupfs используется три аллокатора: для блоков данных, для indirect блоков и для свободных номеров айнодов. Соответственно, блоки данных и indirect блоки находятся каждый в своем адресном пространстве, то есть индексация блоков – своя для каждого аллокатора.

Аллокатор блоков данных реализован с использованием битовой карты, а indirect блоков – как список свободных блоков. Для айнодов не поддерживается никакой дополнительной структуры, а происходит последовательный поиск свободного номера

в таблице айнодов. Данный метод может оказаться неэффективным при большом количестве файлов в директории, однако в нашем случае, учитывая ограничение на максимальное количество файлов, он работает за приемлемое время. Плюсом данного подхода является ненадобность хранения дополнительной структуры на диске, что экономит используемое на служебную информацию дисковое пространство.

3.3. Создание файла

Создание файла происходит следующим образом:

1. Аллокация айнода
2. Запись айнода в хранилище айнодов
3. Аллокация triple-indirect блока
4. Добавление файловой записи в корневую директорию

Первым этапом является аллокация айнода и присваивание ему первого доступного номера. Затем происходит запись айнода в хранилище айнодов. Размер айнода в нашем случае получился 16 байт, поэтому, учитывая, что максимально может быть 128 файлов, для хранилища айнодов вполне хватает одного блока диска. Последним этапом является добавление пары (имя файла, номер айнода) в блок данных корневой директории.

3.4. Чтение файла

Чтение файла с диска происходит следующим образом:

1. Разбиение операции на поблочное чтение
2. Перевод логического блока файла в физический адрес блока
3. Копирование соответствующих блоков данных в userspace

Первым этапом является разделение операции чтения на несколько операций чтения соответствующих логических блоков файла с соответсвующим смещением (offset) и длиной требуемых данных в конкретном блоке.

Затем номер логического блока файла переводится в номер физического блока на диске путем просмотра дисковой структуры файла, начиная от triple-indirect блока и заканчивая ссылкой на требуемый физический блок данных.

Последним этапом является копирование данных из физического блока данных в userspace буфер.

Особенность чтения состоит в том, что мы можем читать несуществующие блоки данных, то есть те логические блоки файла, под которые не было аллоцировано соответствующего физического блока. При попытке чтения такого логического блока буфер заполняется по умолчанию нулевыми байтами. Эта особенность может быть полезна при хранении больших бинарных файлов, данные которых разряжены, то есть существуют большие интервалы между значимой хранящейся информацией, а промежутки между ними заполнены нулевыми байтами. В таком случае у нас появляется возможность хранить на диске только блоки, содержащие значимую информацию, сохраняя тем самым используемое дисковое пространство.

3.5. Запись в файл

Некоторые файловые системы, такие как ext2 и ext3, при записи в файл ищут непрерывную последовательность свободных блоков (например, из 8 блоков) и аллоцируют их. Таким образом файловая система гарантирует, что части файла будут занимать непрерывное пространство на диске, тем самым увеличивая производительность. Такой подход называется pre-allocation policy и часто встречается в промышленных файловых системах. Однако этот метод преаллокации не подходит в нашем случае, поскольку при записи в файл не всегда требуется выделять новые блоки данных под записываемые данные.

Операция записи данных, как и в случае чтения, первоначально разбивается на операции поблочной записи, то есть определяются логические блоки данных файла, куда будет производиться запись.

Далее возможно два варианта: логический блок файла уже существует или же его пока нет, и мы его создаем.

В первом случае, для сохранения консистентности блоков данных, содержимое физического блока копируется в отдельный буфер, имеющий размер блока, до нужного смещения, затем туда же происходит запись новых данных.

Во втором случае происходит копирование новых данных в буфер, а оставшиеся байты заполняются нулевыми значениями. Это делается для того, чтобы далее подсчитанные хэш-суммы были идентичны для эквивалентных неполностью заполненных блоков.

Следующим этапом является расчет двух CRC-16 сумм с различными полиномами от полученного буфера. Выбор значений полиномов для уменьшения числа коллизий является отдельным направлением исследований и не рассматривается в данной работе. Было решено использовать полином 0xa001 для первой хэш-суммы и 0x8005 – для второй. Механизм расчета CRC-16 сумм является распространенной операцией, поэтому не приводится в данной работе.

В нашей файловой системе используется хэш-таблица с цепочками. Выбор CRC-16

хэш-функции обусловлен следующими факторами:

- Отсутствие лавинного эффекта
- Компактное хранение хэш-таблицы на диске

Так как у данной криптографической функции отсутствует лавинный эффект, то при пересчете хэш-функции от измененного блока нам нет необходимости считать новое значение от всего блока данных, а можно ограничиться пересчетом старого значения в зависимости от измененных байт. Это возможно, поскольку измененные байты являются непрерывной последовательностью.

Для хранения хэш-таблицы в случае использования CRC-16 алгоритма необходимо 2^{16} адресов. Если предположить ограничение количества блоков данных размера 4КБ числом 2^{32}, то возможный размер диска ограничивается $2^{32} \cdot 4KB = 16TB$, что является вполне допустимым значением в рабочих условиях. Поэтому для хранения адреса первого блока цепочки коллизий было выделено 4 байта. Таким образом общее место, занимаемое хэш-таблицей на диске, равно $2^{16} \cdot 4B = 128MB$.

После расчета хэш-значений от буфера данных происходит поиск уже хранящегося на диске эквивалентного блока данных в цепочке коллизий, соответствующей первому CRC значению. Для каждого блока данных на диске хранится отдельная структура, обозначим ее кластером. Кластер содержит 4 поля: следующий и предыдущий номера блоков данных в цепочке коллизий, две CRC-16 суммы, посчитанные от данных, хранящихся в соответствующем блоке данных. Для быстрого поиска эквивалентного блока сначала сравниваются значения второй подсчитанной CRC-суммы, и в случае совпадения производится побайтовое сравнение.

Если эквивалентный блок найден, и записываемый логический блок файла уже существовал, то производится уменьшение счетчика ссылок на кластер, соответствующий физическому блоку данных. Если счетчик достигает нуля, то производится удаление старого блока из соответствующей цепочки коллизий, и аллокатор блоков данных освобождает этот блок для дальнейшего использования.

После этого ссылка логического блока файла перенаправляется на найденный эквивалентный блок, счетчик использования соответствующего кластера увеличивается на 1.

В случае, когда эквивалентный блок данных не был найден, и до этого логический блок файла не существовал, необходимо аллоцировать новый блок данных, записать в него данные буфера и добавить кластер в соответствующую цепочку коллизий.

Если же эквивалентный блок данных не был найден, и до этого логический блок данных существовал, то необходимо проверить, можно ли переиспользовать существующий блок данных, то есть переписать старые данные на новые. Это осуществляется просмотром значения счетчика ссылок на кластер: если он равен 1, то мы можем

переиспользовать его, предварительно удалив соответствующую запись из цепочки коллизий, если же счетчик больше единицы, то мы не можем перезаписывать данный блок, поскольку другие файлы ссылаются на него. В данном случае происходит аллоцирование нового блока данных, копирование в него данных из буфера, уменьшение счетчика ссылок на кластер предыдущего блока и перенаправлением ссылки логического блока файла на новый блок.

3.6. Удаление файла

Удаление файла происходит следующим образом: сначала просматриваются ссылки в triple-indirect блоке, затем рекурсивно в низлежайших indirect блоках и так далее. Когда просматриваются кластеры блоков данных, то производится уменьшение счетчика ссылок соответсвующего кластера на 1. Если счетчик ссылок достигает нуля, то данный блок освобождается. Все indirect блоки после просмотра низлежайших блоков также освобождаются.

4. Апробация

4.1. mkfs

Для тестирования функциональности файловой системы во время разработки, а также в общем для использования написанного драйвера файловой системы, необходимо было реализовать userspace утилиту для создания дисковой разметки dedupfs на блочном устройстве.

Поскольку разработка файловой системы ведется на языке C, и все структуры файловой системы описаны в соответствующих заголовочных файлах, то и утилита была написана на языке C.

4.2. proc директория

Для тестирования функционирования монтированной файловой системы был необходим доступ к текущему внутреннему состоянию файловой системы и вывод требуемой статистики для сторонних userspace программ. Поэтому было решено использовать механизм proc директорий ядра Linux.

При регистрации в операционной системе драйвера dedupfs регистрируется папка $/proc/dedupfs$. Так как драйвер может обслуживать несколько точек монтирования, то при монтировании диска с дисковой разметкой dedupfs регистрируется папка /proc/dedupfs/mp<N>, где <N> – число, идентифицирующее очередную точку монтирования. Нумерация начинается с 1.

В папке /proc/dedupfs/mp<N> создаются два файла: *stats* и crc_map.

При чтении файла *stats* выводится список пар для всех файлов, находящихся в хранилище айнодов: номер айнода и размер соответствующего файла в байтах, затем выводится количество занятых блоков данных, а также количество занятых indirect блоков.

Пример вывода файла *stats*

root_dir_inodes: 1
2 654311424
data_blocks: 160092
tf_blocks: 158

При чтении файла crc_map выводится 2^{16} целых беззнаковых чисел, каждое из которых обозначает длину соответствующей цепочки коллизий в хэш-таблице.

4.3. Последовательная запись

Для тестирования скорости последовательной записи было решено произвести эксперименты по записи бинарных файлов разных размеров, содержащих только нулевые байты. Для создания данных файлов, а также измерения времени записи использовалась утилита dd, которая входит в большинство распространенных дистрибутивов Linux.

size, GB	ext4 speed, Mb/s	dedupfs speed, Mb/s
2	176.1	128
3	90.9	125.8
4	78.7	128
5	91.1	126.9

Таблица 1

В таблице 1 приводится среднее время и скорость записи файлов разных размеров, которые содержат только нулевые байты. Измерения проводились на дистрибутиве Arch Linux с версией ядра 4.5 на файловых системах ext4 и dedupfs, соответственно.

Скорость записи в нашей файловой системе оказалась одного порядка, что и в ext4, и даже показала характеристики немного лучше по сравнению с ext4 при больших размерах файлов. Однако, как и следовало ожидать, данный бинарный файл на дисковом пространстве занимает только 4KB, то есть размер одного блока диска, вместо 2GB, 3GB, 4GB и 5GB соответственно.

Пример вывода /proc/dedupfs/mp<N>/stats

root_dir_inodes: 1
2 2147483648
data_blocks: 2
tf_blocks: 514

Следующим тестом было решено произвести запись произвольных байт, считанных из псевдо-устройства /dev/urandom, в файл. В данном случае также использовалась утилита dd.

size, GB	ext4 speed, Mb/s	dedupfs speed, Mb/s
1	18.7	11.4
2	18.8	9.1
3	18.8	9.3

Таблица 2

В таблице 2 приводится среднее время и скорость записи файлов разных размеров, которые содержат произвольные байты, считанные из псевдоустройства /dev/urandom. Измерения проведены на файловых системах ext4 и dedupfs соответственно.

4.4. Произвольная запись

Для измерения скорости произвольной записи было решено написать специальную консольную утилиту, которая открывала файл при помощи системного вызова open() и производила следующие операции в цикле

1. произвольное смещение внутри файла при помощи системного вызова lseek()
2. чтение произвольных байт в буффер из псевдоустройства /dev/urandom
3. запись буффера по заданному смещению в файле при помощи системного вызова write()

Параметры имя файла, размер буффера, а также количество производимых итераций можно было задавать через командные аргументы утилиты.

Результатом работы утилиты являлось время, затраченное на выполнение заданных итераций. Время измерялось при помощи системного вызова clock().

count, 10^5	ext4 speed, Mb/s	dedupfs speed, Mb/s
1	89.75	49.02
2	45.58	22.49
3	29.92	14.75
4	22.39	11.87

Таблица 3

В таблице 3 приводится вычисленная средняя скорость записи в зависимости от количества итераций при размере буффера 2KB.

count, 10^5	ext4 speed, Mb/s	dedupfs speed, Mb/s
1	45.96	26.39
2	22.83	13.62
3	15.12	11.45
4	11.39	10.78

Таблица 4

В таблице 4 приводится вычисленная средняя скорость записи в зависимости от количества итераций при размере буффера 4KB.

4.5. Тест консистентности

Прежде чем переходить к тестированию работы образов виртуальных машин на разработанной файловой системе, необходимо было убедиться, что данные, копируемые на нашу файловую систему, остаются консистентными, то есть при копировании содержимое файлов остается прежним.

Для достижения этой цели были произведены тесты, состоящие из следующей последовательности действий

1. создание файла размером 1GB, состоящего из рандомных байт
2. вычисление md5 суммы от файла, а также измерение времени вычисления
3. копирование созданного файла на dedupfs
4. повторное вычисление md5 суммы от созданного файла, а также измерение времени вычисления
5. сравнение вычисленных значений

В общей сложности было произведено 15 вышеуказанных тестов. Все они показали совпадения вычисленных md5 сумм.

4.6. Тестирование QEMU образов

Для тестирования работы виртуальных машин на носителях с файловой системой dedupfs было решено произвести следующую последовательность действий:

1. создание файла image размером 16GB
2. нанесение дисковой разметки dedupfs на созданный файл image
3. создание файла размером 5GB, состоящего из нулевых байт, и создание на нем QEMU образа Windows XP
4. копирование 3 одинаковых QEMU образов Windows XP на dedupfs
5. Одновременный запуск 3 виртуальных машин
6. Мониторинг потребления дискового пространства

Аналогичная последовательность действий производилась и для файловой системы ext4 кроме первых двух пунктов, поскольку на системе, на которой производилось тестирование, а именно дистрибутив Arch Linux с версией ядра 4.5, ext4 являлась файловой системой для разделов диска, монтированных для корневой и домашней директории.

После запуска виртуальных машин было решено производить мониторинг потребляемого дискового пространства, не производя никаких действий в системе, до тех пор, пока график потребления не стабилизируется.

Затем в каждой из виртуальной машин производилась следующая последовательность действий:

1. Открытие приложения **Калькулятор**
2. Настройка сети
3. Открытие браузера **Internet Explorer**
4. Ожидание полной загрузки стартовой страницы
5. Закрытие браузера
6. Закрытие приложения **Калькулятор**

В случае ext4 файловой системы 3 одинаковых QEMU образа Windows XP занимают 15GB в начальном состоянии и это значение сохраняется во время всей дальнейшей работы виртуальных машин.

Один скопированный QEMU образ Windows XP размером 5GB занимает на dedupfs 222600 блоков по 4KB, то есть 890400 KB.

После копирования последующих двух одинаковых QEMU образов Windows XP на dedupfs количество выделенных под данные блоки остается неизменным, что и следовало ожидать.

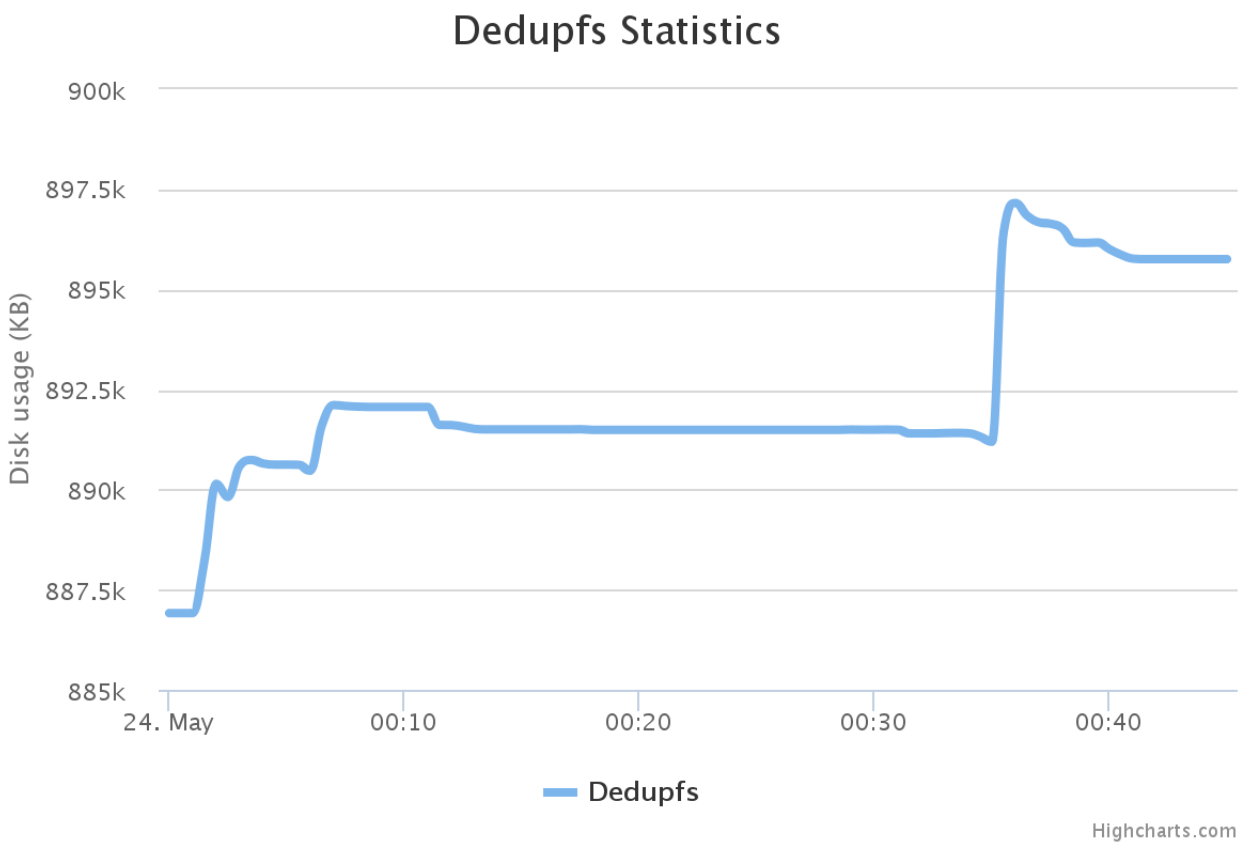

Рис. 5: График потребления дискового пространства для dedupfs

На рис. 5 приведен график потребления дискового пространства от времени при работе 3 виртуальных машин Windows XP на dedupfs. Момент пика графика, зафиксированный на 35 минуте, соответсвует моменту полной загрузки веб-страницы в браузере, далее происходило закрытие открытых приложений.

Как можно заметить, наблюдаются периоды уменьшения занимаемого дискового пространства во время работы виртуальных машин. Данный факт подтвеждает, что разработанная файловая система dedupfs обладает свойством inline дедупликации данных.

5. Результаты

В рамках данной работы были достигнуты следующие результаты.

1. Изучены интерфейсы VFS подсистемы ядра Linux
2. Разработана архитектура файловой системы: спроектированы дисковая разметка и используемые структуры файловой системы для дедупликации данных, а также операции над ними
3. Реализована userspace утилита mkfs для создания дисковой разметки спроектированной файловой системы на блочном устройстве
4. Реализован драйвер файловой системы с поддержкой дедупликации данных на лету для ядра Linux
5. Проведена апробация реализованной функциональности:
    - реализован драйвер, обеспечивающий доступ к внутренней статистике использования смонтированной файловой системы с помощью механизма /proc директорий;
    - проведено сравнение различных подходов к организации процесса дедупликации данных для файловых систем.

Список литературы

[1] *ZFS On-Disk Specification.* Sun microsystems, 2006.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces.* Filesystem Implementation. 2015.

[3] Daniel P. Bovet. *Understanding the Linux Kernel, Third Edition.* The Virtual Filesystem. 2005.

[4] Robert Love. *Linux Kernel Development (3rd Edition).* The Virtual Filesystem. Addison-Wesley Professional, 2010.

[5] Wolfgang Mauerer. *Professional Linux Kernel Architecture.* The Virtual Filesystem. 2008.

[6] Chris Mason Ohad Rodeh, Josef Bacik. *BTRFS: The Linux B-tree Filesystem.* IBM Research, 2012.