Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра электронных вычислительных машин
Дисциплина: Операционные системы и системное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему
УТИЛИТА ПРОВЕРКИ ЦЕЛОСТНОСТИ ФАЙЛОВОЙ СИСТЕМЫ EXT4

БГУИР КП 1–40 02 01 202 ПЗ

Студент: гр. 250502,
Бекетова М. А.

Руководитель:
Басак Д. В.

МИНСК 2024

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра электронных вычислительных машин
Специальность: 40 02 01 «Вычислительные машины, системы и сети»

УТВЕРЖДАЮ
Заведующий кафедрой ЭВМ
_____ Б.В. Никульшин
(подпись)
_____2024 г.

ЗАДАНИЕ
по курсовому проекту студенту
Бекетовой Марии Александровне

1 Тема проекта: «Утилита проверки целостности файловой системы ext4»

2 Срок сдачи студентом законченного проекта: 2 мая 2024 г.

3 Исходные данные к проекту:
3.1 Язык программирования: C.
3.2 Среда разработки: CLion.
3.3 Операционная система: Linux.

4 Содержание пояснительной записки (перечень подлежащих разработке вопросов):
Введение 1. Обзор литературы. 1.1. Обзор существующих аналогов. 1.2. Постановка задачи. 1.3. Выбор языка 1.4. Описание файловой системы ext4. 2. Структурное проектирование. 2.1 Описание деления проекта. 3. Системное проектирование. 4. Функциональное проектирование. 4.1. Описание используемых структур. 4.2. Описание используемых функций. 5. Разработка программных модулей. 6. Руководство пользователя и результаты работы программы. Заключение. Список использованных источников. Приложения.

5 Перечень графического материала (с точным указанием обязательных чертежей):

5.1 Схема структурная

5.2 Диаграмма последовательности

КАЛЕНДАРНЫЙ ПЛАН

Наименование этапов курсового проекта	Объем этапа, %	Срок выполнения этапа	Примечания
Подбор и изучение литературы. Сравнение аналогов.	10	23.03 – 05.04	
Структурное проектирование	15	05.04 – 12.04	
Функциональное проектирование	25	12.04 – 24.04	
Разработка программных модулей	20	24.04 – 08.05	
Программа и методика испытаний	10	8.05 – 15.05	
Оформление пояснительной записки	15	20.05 – 30.05	

Дата выдачи задания: 22.02.2024 г.

РУКОВОДИТЕЛЬ *Басак Д. В.*

(подпись)

Задание принял к исполнению *Бекетова М. А.*

(дата и подпись студента)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Linux (Линукс) – это операционная система, которая на сегодняшний день является фактически единственной альтернативной заменой ОС Windows от Microsoft.

Свое начало Linux берет с 1991 года, когда молодой программист из Финляндии Линус Торвальдс взялся за работу над самой первой версией системы, которая и была названа в честь его имени. Рассвет популярности Linux начался с самого её возникновения. Это связано, в первую очередь, с тем, что ядро этой ОС, как и большинство программ, написанных под нее, обладают очень важными качествами. Пожалуй, главными достоинствами Linux являются безопасность, надежность, бесплатное распространение и открытый исходный код, что и привлекает тысячи программистов по всему миру. Именно благодаря тому, что каждый может внести свои изменения в кодировку, данная ОС пестрит разнообразием.

Разнообразие кроется и в таком понятии как «файловая система». С ним приходится сталкиваться каждому рядовому пользователю, пускай и не так часто, как они об этом задумываются. Возникает логичный вопрос: что такое файловая система и какие они бывают?

Файловая система – это инструмент, позволяющий операционной системе и программам обращаться к нужным файлам и работать с ними. При этом программы оперируют только названием файла, его размером и датой создания. Все остальные функции по поиску необходимого файла в хранилище и работе с ним берет на себя файловая система накопителя.

В отличие от ОС Windows и macOS, ограничивающих выбор файловой системы предустановленными вариантами, Linux предоставляет возможность использования нескольких файловых систем, каждая из которых оптимизирована для решения определенных задач. Файловые системы в Linux используются не только для работы с файлами на диске, но и для хранения данных в оперативной памяти или доступа к конфигурации ядра во время работы системы. Все они включены в ядро и могут использоваться в качестве корневой файловой системы.

Основные файловые системы, используемые в дистрибутивах Linux:

- Ext2;
- Ext3;
- Ext4;
- JFS;
- ReiserFS;
- XFS;
- ZFS.

Linux поддерживает деление жесткого диска на разделы. Для подсчета и определения физических границ используется специальная таблица разделов — GPT или MBR. Она содержит метку и номер раздела, а также адреса физического расположения точек начала и конца раздела.

В Linux на каждый раздел можно установить свою файловую систему, которая отвечает за порядок и способ организации информации. В основе файловых систем лежит набор правил, определяющий, где и каким образом хранятся данные. Следующий «слой» файловой системы — практический (технический) способ организации информации на каждом конкретном типе носителя (опять же, учитывая правила, заложенные в основу системы).

От выбора файловой системы зависят:

- скорость работы с файлами;
- их сохранность;
- скорость записи;
- размер файлов.

Тип файловой системы также определяет, будут ли данные храниться в оперативной памяти (ОП) и как именно пользователь сможет изменить конфигурацию ядра.

Файловая система (ФС) — архитектура хранения данных, которые могут находиться в разделах жесткого диска и ОП. Она выдает пользователю доступ к конфигурации ядра. Определяет, какую структуру принимают файлы в каждом из разделов, создает правила для их генерации, а также управляет файлами в соответствии с особенностями каждой конкретной ФС.

ФС Linux — пространство раздела, поделенное на блоки определенного размера. Он определяется кратностью размеру сектора. Соответственно, это могут быть 1024, 2048, 4096 или 8120 байт. Важно помнить, что размер каждого блока известен изначально, ограничен максимальным размером ФС и зависит от требований, которые выдвигает пользователь к каждому из блоков.

Для обмена данными существует сразу два способа. Первый из них — виртуальная файловая система (VFS). С помощью данного типа ФС происходит совместная работа ядра и приложений, установленных в системе. VFS позволяет пользователю работать, не учитывая особенности каждой конкретной ФС. Второй способ — драйверы файловых систем. Именно они отвечают за связь между «железом» и программным обеспечением. Список файловых систем, которые поддерживаются ядром, находится в файле `/proc/filesystems`.

Файловая система в Linux определяет также организацию расположения файлов, по сути, представляя собой иерархическую структуру «дерева»: начинается с корневого каталога «/» и разрастается ветвями в зависимости от работы системы.

ФС также характерно понятие целостности: в такой системе изменения, внесенные в один файл, не приведут к изменению другого файла, не связанного с первым. У всех данных есть собственная физическая память. В Linux целостность ФС проверяется специальной командой — `fsck`.

Типы файлов условно можно разделить на несколько групп. Некоторые из них такие же, как и в ОС Windows, — текстовые документы, медиа и изображения. Отличия начинаются с каталогов, которые являются отдельным

типом файлов. Жесткие диски относят к блочным устройствам. Принтеры — к символьным. Отдельную группу составляют символические ссылки, о которых речь пойдет ниже. К типам файлов относится каналы межпроцессного взаимодействия — PIPE (FIFO), а также гнезда (разъемы центрального процессора).

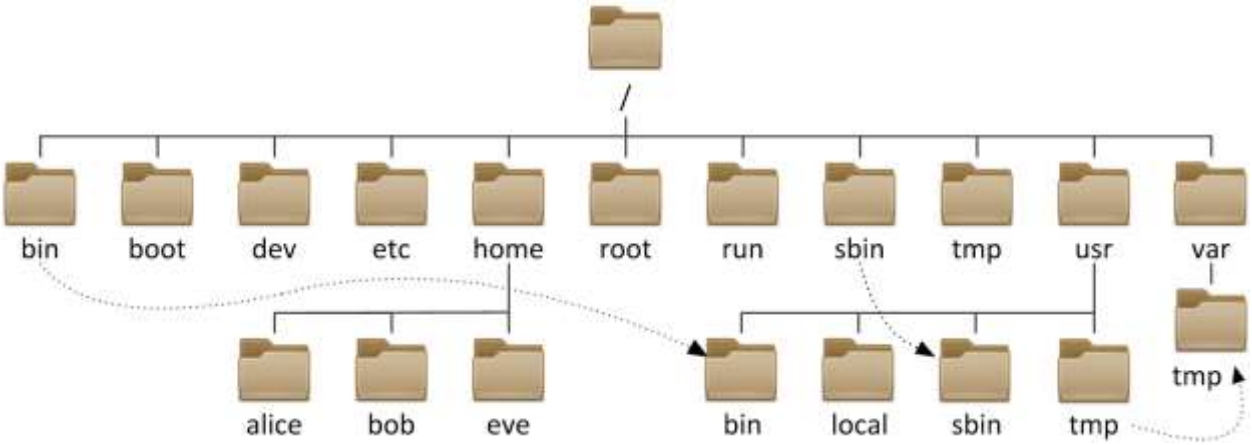

Рисунок 1 – Структура каталогов в Linux

В ФС каждый файл определяется конкретным индексом — Inode (от англ. index node — «индексный дескриптор»). Но при этом один файл (речь о физическом размещении) может иметь сразу несколько имен (или путей). И если в структуре ФС файлы будут отличаться, то на жестком диске им может соответствовать один файл. Это означает, что ФС Linux перекрестно-иерархична, а ветви дерева могут пересекаться.

Корневой раздел в Linux один — «/» (root, «корень)». Разделы называются подкаталогами, примонтированными к соответствующим каталогам.

При монтировании происходит ассоциирование каталога с устройством, содержащим ФС (драйвер). Соответствующая ссылка на устройство передается драйверу. Именно он и определяет ФС. Если процедура завершается успешно, ядро заносит информацию (каким драйвером обслуживаются и где расположены файлы и каталоги) в таблицу монтирования. Она находится в файле `/proc/mounts`.

Данные о каждом файле содержит Inode — специфичный для UNIX-систем индексный дескриптор, хранящий различную метаинформацию (владелец файла, последнее время обращения, размер и так далее).

Когда файл (каталог) перемещается в другую ФС, его Inode тоже создается заново. И только потом удаляется исходный (в рамках той же системы меняется только путь файла). Также отметим, что файл (каталог) существует до того момента, пока хранится информация о его имени или пути к нему. После удаления всей информации блоки, отведенные под файл, становятся свободными (для выделения под другой файл).

Еще одна особенность Linux: существование сразу двух типов ссылок. Во-первых, жесткая ссылка (Hard-Link), которая представляет собой один из путей файла. Во-вторых, символьная ссылка (Symbolic link) — это файл UNIX с текстовой строкой с путем к оригинальному файлу.

Общая информация о ФС хранится в суперблоке. Сюда относится суммарное число блоков и Inode, число свободных блоков, их размеры и так далее. Важно, чтобы суперблок сохранял свою целостность, поскольку от этого зависит стабильность и работоспособность системы в целом. В ОС создается сразу несколько копий, чтобы можно было восстановить всю необходимую информацию.

При загрузке ядро автоматически монтирует разделы после того, как корень уже примонтирован на чтение. Информацию ядро считывает из конфигурации `/etc/fstab`.

Таким образом, не стоит закрывать глаза на выбор файловой системы, т.к. это является важным шагом к началу использования компьютера.

По мере роста наших потребностей в хранилищах новые функции и организация новых файловых систем становятся все более полезными и необходимыми.

Краткая характеристика популярных систем:

- ext4 – это стандартный и безопасный выбор.
- XFS – также довольно стабильна и отлично подходит для больших файлов и тяжелой многопроцессорной обработки.
- btrfs – гибка и мощна, но имеет недостатки.
- ZFS – хорошо протестирована и достаточно надежна, но более сложна. В обмен на сложность она решает проблемы хранения в более широком масштабе.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Обзор существующих аналогов

Во время выполнения различных задач по администрированию системы может потребоваться работа с файловой системой Linux, включая форматирование разделов, изменение их размера, конвертацию файловых систем, выполнение дефрагментации в Linux или восстановление файловых систем.

Многие из этих действий можно выполнять в графическом интерфейсе, а некоторые даже автоматизированы. Однако могут возникнуть ситуации, когда приходится использовать терминал. Кроме того, при администрировании удаленных серверов приходится работать с ними только через SSH, что означает отсутствие графического интерфейса.

На сегодняшний день самой популярной файловой системой для пользователей Linux является файловая система семейства ext, особенно ext4. Она обладает максимальным набором функций и считается наиболее стабильной благодаря редким изменениям в кодовой базе. Начиная с версии ext3, в системе используется функция журналирования. В настоящее время версия ext4 присутствует во всех дистрибутивах Linux.

Для работы с ext4 существует специальная утилита e2fsck. Это мощный инструмент, предназначенный для проверки файловой системы на наличие ошибок и их исправления.

`e2fsck` используется для проверки второй расширенной файловой системы Linux (`ext2fs`). `E2fsck` также поддерживает файловые системы `ext2`, содержащие журнал, которые также иногда называют файловыми системами `ext3`, сначала применяя журнал к файловой системе, прежде чем продолжить обычную обработку `e2fsck`. После применения журнала файловая система обычно помечается как чистая. Следовательно, для файловых систем `ext3 e2fsck` обычно запускает журнал и завершает работу, если только его суперблок не указывает, что требуется дальнейшая проверка.

Обратите внимание, что в целом небезопасно запускать `e2fsck` в смонтированных файловых системах. Единственным исключением является случай, когда указан параметр `-n`, а параметры `-c, -l` или `-L` не указаны. Однако, даже если это безопасно, результаты, напечатанные `e2fsck`, недействительны, если файловая система смонтирована. Если `e2fsck` спросит, следует ли вам проверять смонтированную файловую систему, единственный правильный ответ – нет. Только эксперты, которые действительно знают, что они делают, должны рассмотреть возможность ответа на этот вопрос каким-либо другим способом.

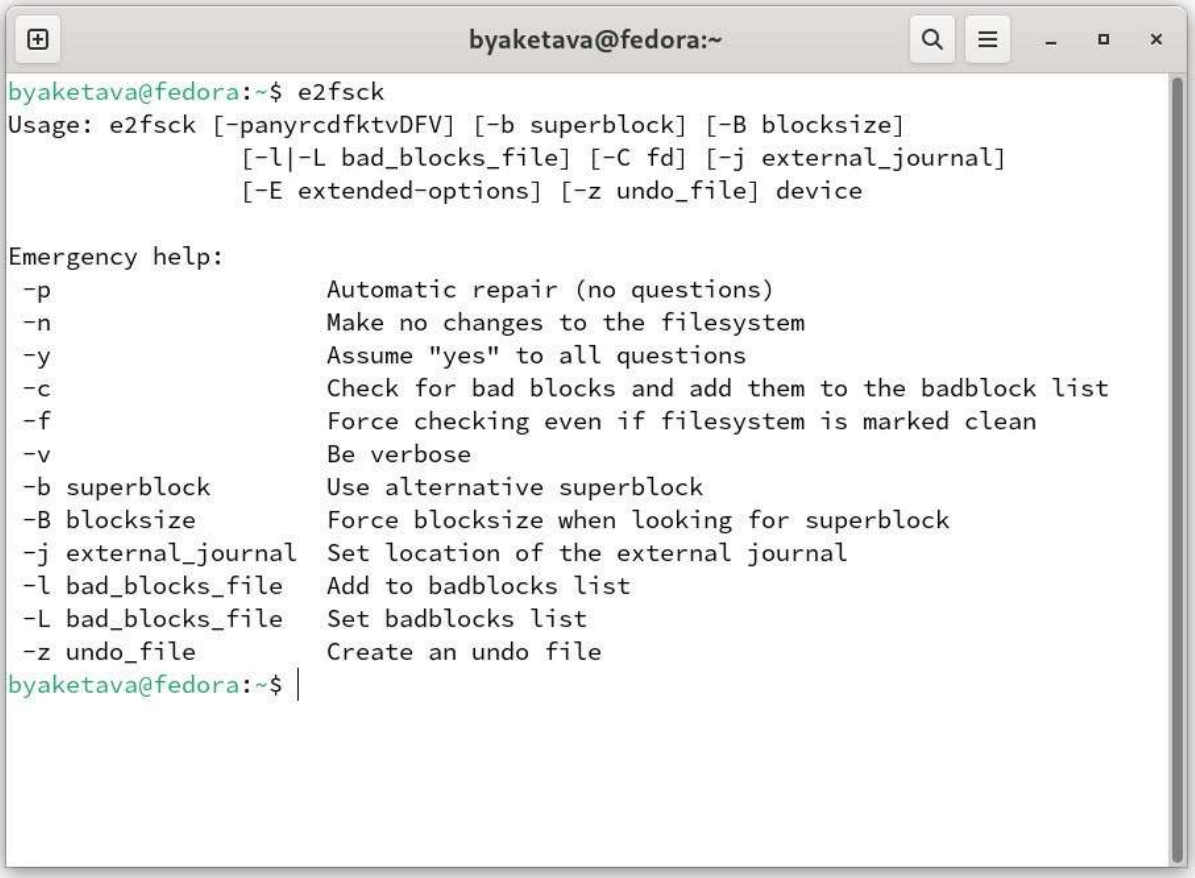

Рисунок 1.1 – Утилита e2fsck

Особенностью данной утилиты является обширный функционал, предоставляющий не только возможность проверки целостности файловой системы, но и автоматическое исправление ошибок.

Кроме `e2fsck` существует и общая для всех файловых систем утилита `fsck`. `Fsck` расшифровывается как «проверка целостности файловой системы» (file system consistency check). Основная её функция заключается в восстановлении логической непротиворечивости файловой системы, созданной в разделе жесткого диска.

Эта утилита по умолчанию входит в состав дистрибутивов Linux. Для использования `fsck` не требуется никаких специальных шагов или процедуры установки. Однако, запускать её необходимо с привилегиями суперпользователя. Использовать можно с разными аргументами. Их использование зависит от вашего конкретного случая.

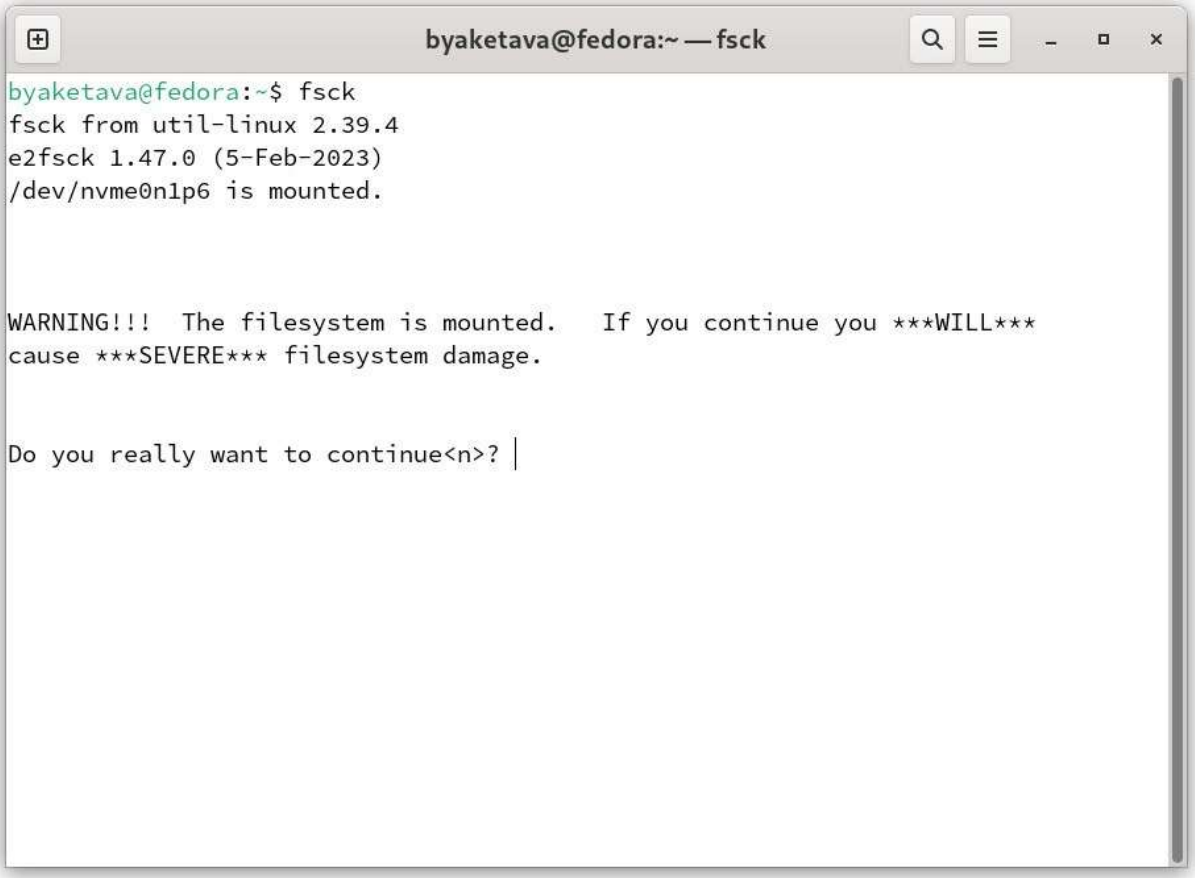

Рисунок 1.2 – Утилита fsck

Утилита не будет работать на смонтированных разделах, так как это может привести к повреждению файловой системы, поэтому перед использованием `fsck` необходимо сделать размонтирование раздела с помощью команды `unmount`.

`mkfs` используется для создания файловой системы Linux на некотором устройстве, обычно в разделе жёсткого диска. В качестве аргумента `filesys` для файловой системы может выступать или название устройства или точка монтирования. Аргументом `blocks` указывается количество блоков, которые выделяются для использования этой файловой системой.

По окончании работы `mkfs` возвращает 0 - в случае успеха, а 1 - при неудачной операции.

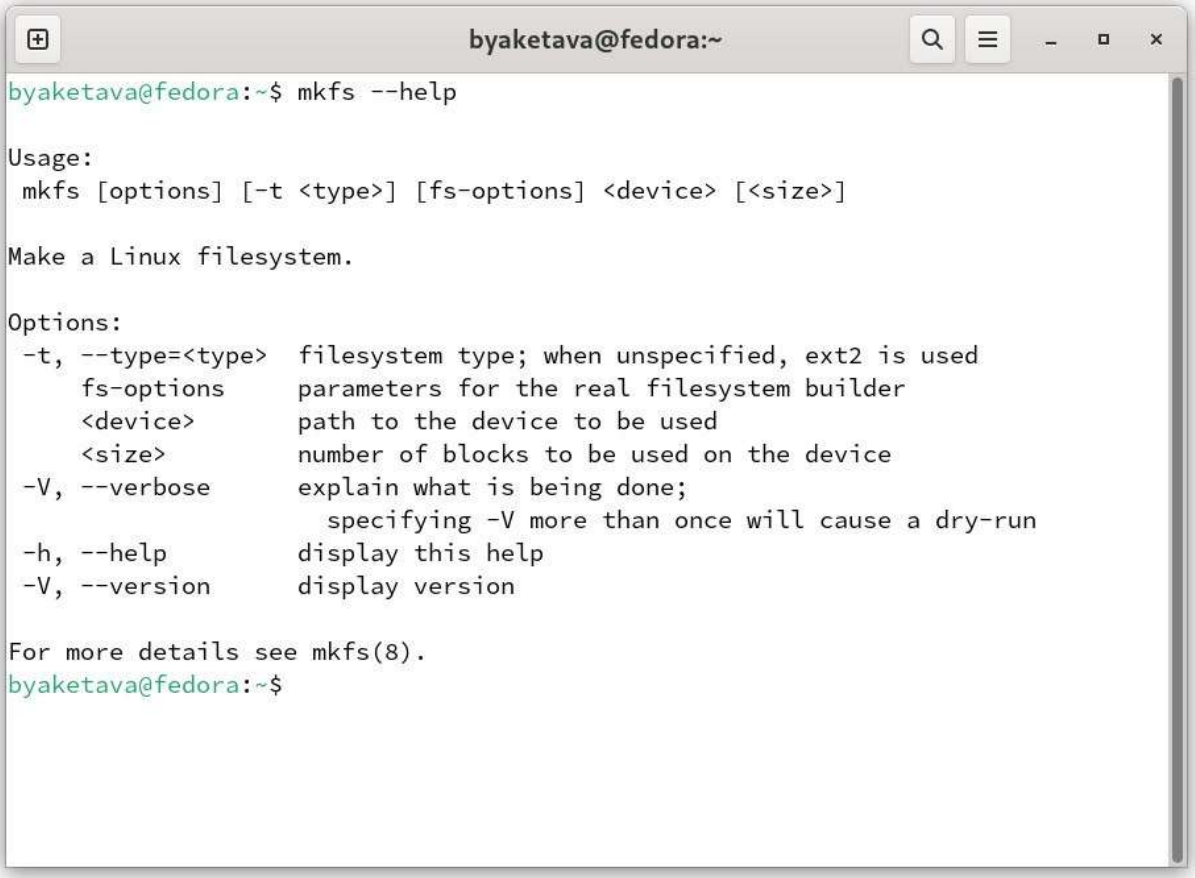

Рисунок 1.3 – Утилита mkfs

В общем случае, `mkfs` является простым конечным интерфейсом к доступным под Linux модулям создания файловых систем, в которых вторая часть сложных имён (`mkfs.fstype`) как раз и определяет вызываемый модуль. Точный список каталогов определяется во время компиляции, но как минимум содержит `/sbin` и `/sbin/fs`, а завершается каталогами, которые перечислены в переменной окружения PATH.

Различные параметры файловой системы, такие как размер блока данных, иноды или зарезервированное место под данные пользователя root можно настроить. Для этого существует утилита `tune2fs`.

`tune2fs` позволяет вам изменять разные характеристики файловых систем. А еще вы сможете увидеть, какие параметры уже установлены. Просмотр текущих характеристик файловой системы с помощью `tune2fs`. Команда `tune2fs -l` покажет вам всю информацию, которая держится в суперблоке файловой системы.

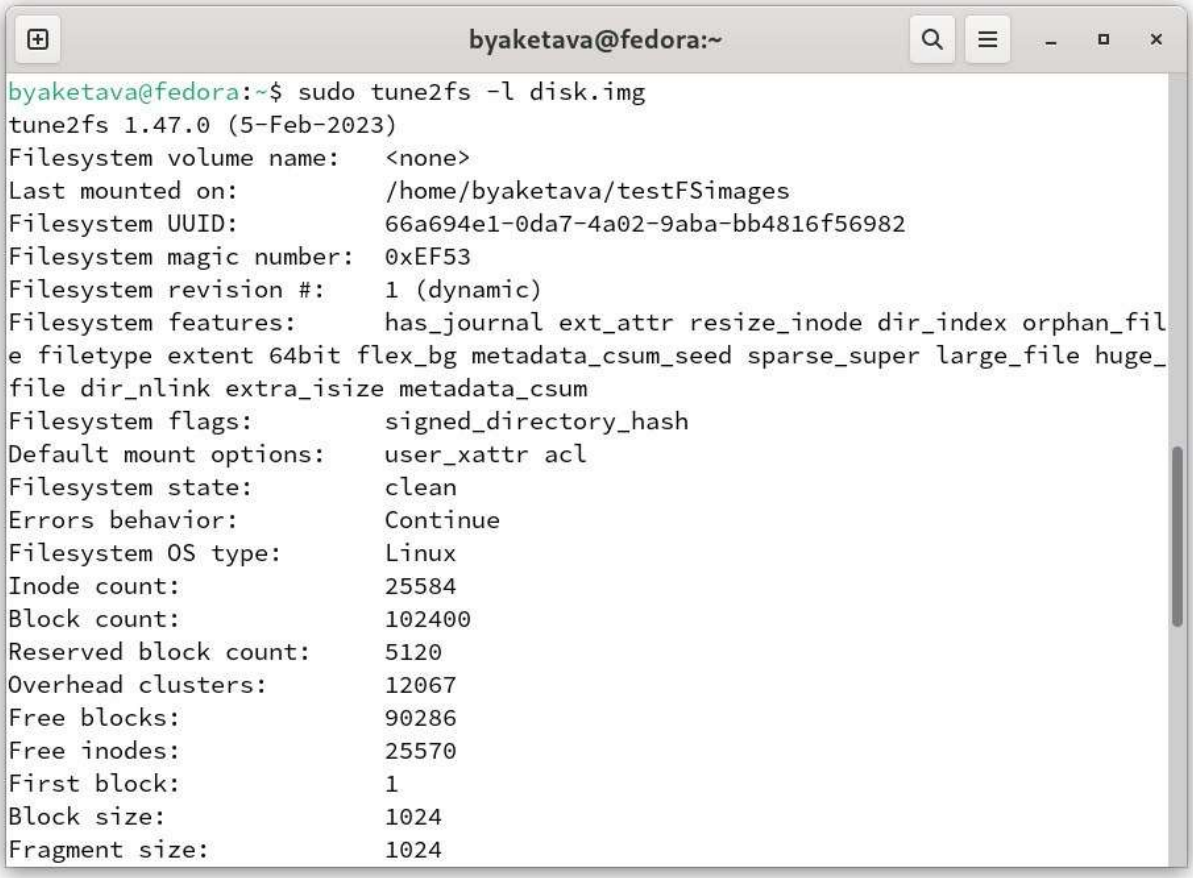

Рисунок 1.4.—Утилита tune2fs

Изменение размера или расширение раздела – это процесс увеличения или уменьшения размера раздела. Вы можете либо расширить, либо уменьшить раздел в соответствии с вашими потребностями. Вам нужно будет изменить размер раздела, если вас не устраивает размер раздела, в вашем системном разделе заканчивается свободное место, какой-то раздел заполнен, а на каком-то разделе все еще много свободного места.

Будучи пользователем системы Linux, изменение размера или расширение файловой системы является довольно сложной задачей. Нужно будет изменить размер существующего раздела, когда ваш размер раздела будет заполнен. В этом случае можно использовать утилиту `resize2fs` для увеличения и уменьшения размера файловой системы. `resize2fs` – это утилита командной строки, которая позволяет изменять размер файловых систем ext2, ext3 или ext4.

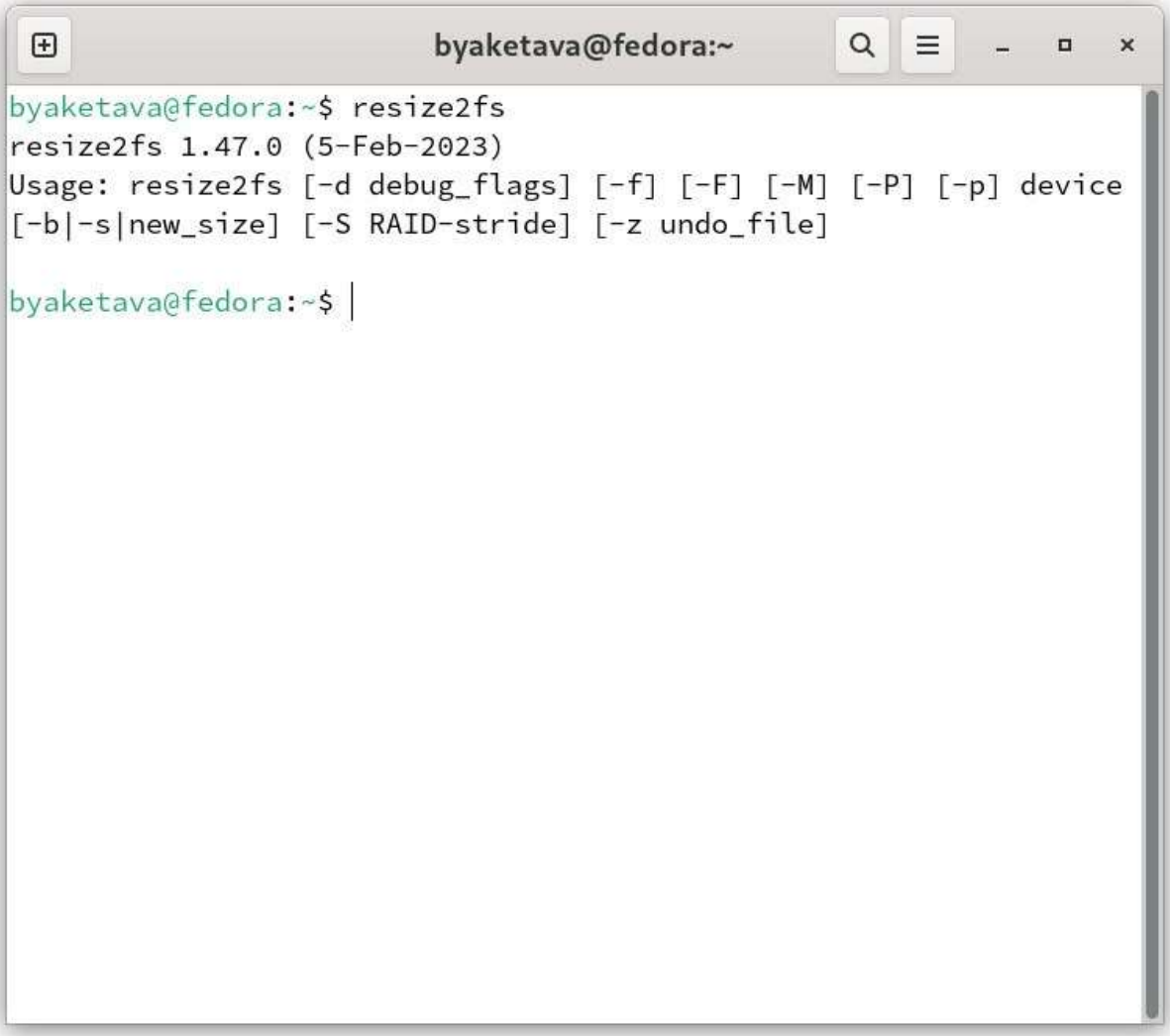

Рисунок 1.5 – resize2fs

1.2 Постановка задачи

После ознакомления с существующими аналогами, был выделен ряд основных требований, которые должны быть выполнены при реализации курсовой работы.

Пользователь утилиты должен иметь возможность проверить целостность файловой системы путём прочтения её образа и определением согласованности ряда вещей. Требуется учитывать все ошибки, которые могут появиться в ходе проверки.

Программа должна иметь удобный пользовательский интерфейс, позволяющий в полной мере использовать весь функционал программы. При обнаружении ошибки должно выводится соответствующее сообщение.

1.3 Выбор языка

C – это достаточно «древний» язык программирования, он сформировался еще в начале 70-х. Несмотря на это, C — живой язык в том смысле, что он активно применяется в настоящее время. Он был придуман, использовался и используется для написания частей программного кода Unix-подобных операционных систем. Также на нем пишут утилиты, компиляторы и реже прикладные программы. Поэтому C называют системным языком программирования.

Его живучесть можно объяснить тем, что принципы работы операционных систем относительно универсальны, они не подвержены тому прогрессу и разнообразию, которые можно наблюдать в среде десктопного и мобильного ПО, Web-приложений. C не является языком достаточно высокого уровня, он ближе к архитектуре компьютера. В результате программы на C получаются более быстрыми.

C в чистом виде не поддерживает объектно-ориентированного программирования (хотя есть библиотека, в которой эмулируются возможности ООП). Поддержка ООП реализована в C++. Хотя последний возник на основе языка C, он не является его "продолжением", а представляет собой отдельный язык, который можно изучать, не зная C. Однако изучение C полезно перед знакомством с его "продвинутым младшим братом", т.к. синтаксис языков похож, C не перегружает мозг начинающего программиста сверхвозможностями и приучает к пониманию сути происходящего.

C – компилируемый язык программирования. В GNU/Linux для получения исполняемых файлов используется GCC – набор компиляторов, включающий в том числе компилятор для C.

Программа, написанная на языке C, состоит из операторов. Каждый оператор вызывает выполнение некоторых действий на соответствующем шаге выполнения программы.

При написании операторов применяются латинские прописные и строчные буквы, цифры и специальные знаки. К таким знакам, например, относятся: точка (.), запятая (,), двоеточие (:), точка с запятой (;) и др. Совокупность символов, используемых в языке, называется алфавитом языка.

В персональном компьютере символы хранятся в виде кодов. Соответствие между каждым символом и его кодом задается специальной кодовой таблицей. На нее разработан стандарт ASCII, поэтому коды символов называют ASCII-кодами.

Различают видимые и управляющие символы. Первые могут быть отображены на экране дисплея либо отпечатаны на принтере. Вторые вызывают определенные действия в машине, например: звуковой сигнал - код 710, возврат курсора на один шаг - код 810, горизонтальная табуляция - код 910, перевод курсора на новую строку - код 1010, перемещение курсора в начало строки - код 1310 и т.д. Такие управляющие символы имеют десятичные номера 0 - 31, 127.

Первая половина кодовой таблицы является стандартной, а вторая используется для представления символов национальных алфавитов, псевдографических элементов и т.д.

Важным понятием языка является идентификатор, который используется в качестве имени объекта (функции, переменной, константы и др.). Идентификаторы должны выбираться с учетом следующих правил:

- Они должны начинаться с буквы латинского алфавита (a,...,z, A,...,Z) или с символа подчеркивания (_).
- В них могут использоваться буквы латинского алфавита, символ подчеркивания и цифры (0,...,9). Использование других символов в идентификаторах запрещено.
- В языке Си буквы нижнего регистра (a,...,z), применяемые в идентификаторах, отличаются от букв верхнего регистра (A,...,Z). Это означает, что следующие идентификаторы считаются разными: name, NaMe, NAME и т.д.
- Идентификаторы могут иметь любую длину, но воспринимается и используется для различения объектов (функций, переменных, констант и т.д.) только часть символов. Их число меняется для разных систем программирования, но в соответствии со стандартом ANSI C не превышает 32 (в Си++ это ограничение снято). Если длина идентификатора установлена равной 5, то имена count и counter будут идентичны, поскольку у них совпадают первые пять символов.
- Идентификаторы для новых объектов не должны совпадать с ключевыми словами языка и именами стандартных функций из библиотеки.

В программах на языке Си важная роль отводится комментариям. Они повышают наглядность и удобство чтения программ. Комментарии обрамляются символами /* и */. Их можно записывать в любом месте программы.

В языке C++ введена еще одна форма записи комментариев. Все, что находится после знака // до конца текущей строки, будет также рассматриваться как комментарий. Отметим, что компилятор языка Си, встроенный в систему программирования Borland C++, позволяет использовать данный комментарий и в программах на C.

Пробелы, символы табуляции и перехода на новую строку в программах на Си игнорируются. Это позволяет записывать различные выражения в хорошо читаемом виде. Кроме того, строки программы можно начинать с любой позиции, что дает возможность выделять в тексте группы операторов.

В языке различают понятия "тип данных" и "модификатор типа". Тип данных – это, например, целый, а модификатор - со знаком или без знака. Целое со знаком будет иметь как положительные, так и отрицательные значения, а целое без знака - только положительные значения. В языке Си можно выделить пять базовых типов, которые задаются следующими ключевыми словами:

- char - символьный;
- int - целый;
- float - вещественный;
- double - вещественный двойной точности;
- void - не имеющий значения.

1.4 Описание файловой системы ext4

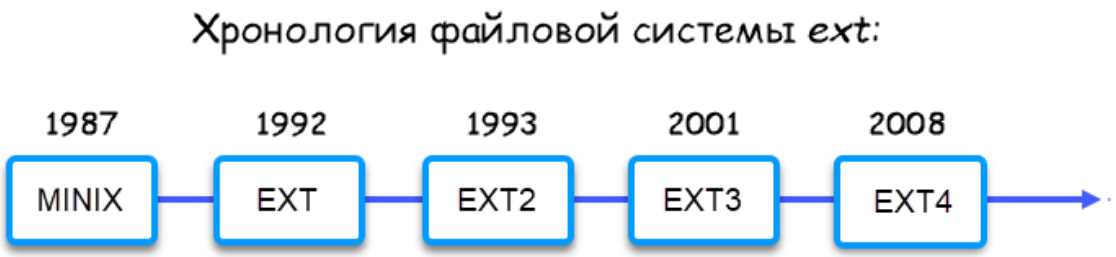

Рисунок 1.6 – Хронология файловой системы ext

Ext4 — это результат эволюции Ext3, наиболее популярной файловой системы в Linux. Во многих аспектах Ext4 представляет собой больший шаг вперёд по сравнению с Ext3, чем Ext3 была по отношению к Ext2. Наиболее значительным усовершенствованием Ext3 по сравнению с Ext2 было журналирование, в то время как Ext4 предполагает изменения в важных структурах данных, например, предназначенных для хранения данных файлов. Это позволило создать файловую систему с более продвинутым дизайном, более производительную и стабильную и с обширным набором функций.

Первая экспериментальная реализация ext4 была написана Эндрю Мортоном и выпущена 10 октября 2006 года в виде патча к ядрам Linux версий 2.6.19-rc1-mm1 и 2.6.19-rc1-git8.

В октябре 2008 была переименована из ext4dev в ext4, что символизирует то, что с точки зрения разработчиков она достаточно стабильна. В ядре 2.6.28 (вышедшем 25.12.2008) файловая система уже называется ext4 и считается стабильной. Файловая система ext4 рассматривается как промежуточный шаг на пути к файловой системе следующего поколения Btrfs, которая претендует на звание основной файловой системы Linux в будущем.

В структуре Ext можно выделить несколько основных составляющих:

- Суперблок – находится в самом начале файловой системы (обычно в первых 1024 байтах раздела). Система автоматически создает несколько копий суперблока, так как без него она не сможет функционировать. В суперблоке хранится базовая информация о файловой системе, а именно:
  - общее число блоков данных и inode для всей файловой системы;
  - количество свободных inode и блоков данных, в которые можно

будет записать файлы;
    - размер inode и блока данных (эти данные указываются при создании файловой системы);
    - информация о файловой системе – время монтирования, последние изменения и т.д.
- Сразу же после суперблока находится глобальная таблица дескрипторов группы блоков (Group Description Table). В ней описаны первый и последний блоки для каждой группы блоков, а также информация, где именно в каждой группе начинается таблица inode, начало блоков данных и т.д.
- Битовая карта блока (Block Bitmap) – это специальная таблица, в которой указано какие блоки в группе использованы, а какие свободны. Эта информация используется во время распределения информации в блоке. 0 – означает что блок свободен, а 1 – что занят.
- Битовая карта inode (Inode Bitmap) – эта таблица аналогична битовой карте блока, только в ней отображается информация о свободных inode, которые могут быть использованы для записи новых файлов.
- Блоки данных – выделенные физические блоки памяти, в которых хранятся данные пользователя.

Основные изменения по сравнению с ext3:

- увеличение максимального объёма одного раздела диска до 1 эксбибайта (260 байт) при размере блока 4 кибибайт;
- увеличение размера одного файла до 16 тебибайт (244 байт);
- введение механизма пространственной (extent) записи файлов,уменьшающего фрагментацию и повышающего производительность. Суть механизма заключается в том, что новая информация добавляется в конец области диска, выделенной заранее по соседству с областью, занятой содержимым файла.

Возможности ext4:

- Использование экстентов. В файловой системе ext3 адресация данных выполнялась традиционным образом, поблочно. Такой способ адресации становится менее эффективным с ростом размера файлов. Экстенты позволяют адресовать большое количество (до 128 MB) последовательно идущих блоков одним дескриптором. До 4х указателей на экстенты может размещаться непосредственно в inode, что достаточно для файлов маленького и среднего размера.
- 48-и битные номера блоков. На сегодняшний день максимальный размер файловой системы ext3 равен 16 терабайтам, а размер файла ограничен 2 терабайтами. В ext4 добавлена 48-битная адресация блоков, что означает, что максимальный размер этой файловой системы равен одному экзабайту, и файлы могут быть размером до 16 терабайт. 1 EB (экзабайт) = 1,048,576 TB (терабайт), 1 EB = 1024 PB (петабайт), 1 PB = 1024 TB, 1 TB = 1024 GB.
- Выделение блоков группами (англ. multiblock allocation). Файловая

система хранит не только информацию о местоположении свободных блоков, но и количество свободных блоков, идущих друг за другом. При выделении места файловая система находит такой фрагмент, в который данные могут быть записаны без фрагментации. Это снижает уровень фрагментации ФС в целом.

- Отложенное выделение блоков (англ. delayed allocation). Выделение блоков для хранения данных файла происходят непосредственно перед физической записью на диск (например, при вызове sync), а не при вызове write. В результате, операции выделения блоков можно делать не по одной, а группами, что в свою очередь минимизирует фрагментацию и ускоряет процесс выделения блоков. С другой стороны, увеличивает риск потери данных в случае внезапного пропадания питания.
- Масштабируемость подкаталогов. В настоящий момент один каталог ext3 не может содержать более, чем 32000 подкаталогов. Ext4 снимает это ограничение и позволяет создавать неограниченное количество подкаталогов.
- Резервирование inode при создании каталога (англ. directory inodes reservation). При создании каталога резервируется несколько inode. Впоследствии, при создании файлов в этом каталоге сначала используются зарезервированные inode, и если таких не осталось, выполняется обычная процедура.
- Размер inode. Размер inode (по умолчанию) увеличен с 128 до 256 байтов. Это дало возможность реализовать те преимущества, которые перечислены ниже.
- Временные метки с наносекундной точностью (англ. nanosecond timestamps). Точность временных меток, хранящихся в inode, повышена до наносекунд. Диапазон значений тоже расширен: у ext3 верхней границей хранимого времени было 18 января 2038 года, а у ext4 — 25 апреля 2514 года.
- Версия inode. В inode появился номер, который увеличивается при каждом изменении inode файла. Это будет использоваться, например, в NFSv4, для того чтобы узнавать, изменился ли файл.
- Хранение расширенных атрибутов в inode (англ. extended attributes (EA) in inode). Хранение расширенных атрибутов, таких как ACL, атрибутов SELinux и прочих, позволяет повысить производительность. Атрибуты, для которых недостаточно места в inode, хранятся в отдельном блоке размером 4KB. Предполагается снять это ограничение в будущем.
- Контрольное суммирование в журнале (Journal checksumming). Контрольные суммы журнальных транзакций. Позволяют лучше найти и (иногда) исправить ошибки при проверке целостности системы после сбоя.
- Режим без журналирования. Журналирование обеспечивает целостность файловой системы путём протоколирования всех происходящих на диске изменений. Но оно также вводит дополнительные накладные

расходы на дисковые операции. В некоторых особых ситуациях журналирование и предоставляемые им преимущества могут оказаться излишними. Ext4 позволяет отключить журналирование, что приводит к небольшому приросту производительности.

- Предварительное выделение (англ. persistent preallocation). Сейчас для того, чтобы приложению гарантированно занять место в файловой системе, оно заполняет его нулями. В ext4 появилась возможность зарезервировать множество блоков для записи и не тратить на инициализацию лишнее время. Если приложение попробует прочитать данные, оно получит сообщение о том, что они не проинициализированы. Таким образом, несанкционированно прочитать удалённые данные не получится.
- Дефрагментация без размонтирования (англ. online defragmentation). Дефрагментация выполняется утилитой e4defrag, поставляемой в составе пакета e2fsprogs с 2011 года.
- Неинициализированные блоки (англ. uninitialised groups). Возможность пока не реализована и предназначена для ускорения проверки целостности ФС утилитой fsck. Блоки, отмеченные как неиспользуемые, будут проверяться группами, и детальная проверка будет производится только если проверка группы показала наличие повреждений. Предполагается, что время проверки будет составлять от 1/2 до 1/10 от нынешнего в зависимости от способа размещения данных.
- Прямая и обратная совместимость с ext2/ext3. Файловые системы ext2/ext3 можно монтировать как файловую систему ext4. Наоборот — монтировать файловую ext4 как ext3 — можно только в том случае, если на ext4 не используются экстенты.

Отсутствие острой необходимости в дефрагментации файловых систем UNIX на фоне регулярной дефрагментации других популярных файловых систем укрепило системных администраторов UNIX во мнении, что фрагментации данных в их файловых системах не бывает в принципе. В действительности, она существует, хотя её влияние и не настолько существенно, как в файловых системах некоторых других архитектур. С ростом объёма файловых систем необходимость борьбы с фрагментацией становится более ощутимой.

Результаты экспериментов показали, что производительность файловых систем UNIX из-за фрагментации может снижаться достаточно сильно.

Например, был проведён следующий эксперимент: на чистую файловую систему было записано 32 файла размером 1GB каждый; сначала по очереди, потом (после пересоздания файловой систем) одновременно. За счёт параллельности процессов записи во втором случае файловая система получилась сильно фрагментированной. Различие в скорости чтения данных в первом и втором случае составило 14.8% для ext3 и 16.5% для XFS.

Виды фрагментации:

- фрагментация отдельных файлов – большой файл занимает блоки,

разбросанные по файловой системе;
- фрагментация связанных файлов – файлы, читающиеся вместе, разбросаны по файловой системе;
- фрагментация свободного места – свободные блоки разбросаны по файловой системе.

Фрагментация может быть снижена при помощи отложенного выделения блоков, резервирования блоков и многоблочного выделения.

Однако стоит понимать, что при использовании SSD вместо HDD фрагментация никак не влияет на производительность операций чтения/записи (для операций записи это корректно при достаточном количестве свободного пространства на разделе и поддержке накопителем опции TRIM).

Файловая система ext4 находится в состоянии развития. Её уже можно использовать для экспериментов, но пока что не рекомендуется хранить на ней ценные данные.

Основные возможности ext4, поддержка которых на сегодняшний день не включена в основной код:
- отложенное выделение блоков;
- online-дефрагментация;
- контрольное суммирование журнала;
- восстановление удалённых файлов.

Среди ограничений в программах (userlevel tools), необходимых для работы с ext4, существующих на сегодняшний день, самым большим является максимальный размер файловой системы – не может превышать 16TB.

Это связано с тем, что существующая mkfs не умеет пока что работать в 64-битном режиме (но поддержка со стороны ядра есть).

В сети все чаще появляется информация, что поскольку ext4 не поддерживает функции следующего поколения – операционная система Linux в ближайшее время перейдёт на btrfs. Поэтому, многие пользователи задаются вопросом стоит ли использовать ext4 или лучше сразу же установить альтернативу в виде btrfs или какой-то другой файловой системы?

Тут каждый принимает решение сам, однако, как ни крути ext4 не поддерживает многие современные функции. Поэтому, если вы хотите использовать файловую систему, проверенную временем, но готовы отказаться от многих современных возможностей – ваш выбор ext4. Она стабильна, широко поддерживается и отлично работает.

Если для вас важно иметь современную файловую систему, которая будет поддерживать все новые функции и в то же время не боитесь иметь дело с несколько менее зрелой экосистемой — ваш выбор btrfs. Кроме того, вы будете на шаг впереди, поскольку в ближайшее время большинство дистрибутивов Linux будет по умолчанию ее использовать.

2 СТРУКТУРНОЕ ПРОЕКТИРОВАНИЕ

2.1 Описание деления проекта

Образ диска (image) – файл, несущий в себе полную копию содержимого и структуры файловой системы и данных, находящихся на диске, таком как компакт-диск, дискета, раздел жёсткого диска или весь жёсткий диск целиком.

Термин описывает любой такой файл, причём неважно, был ли образ получен с реального физического диска или нет. Таким образом, образ диска содержит всю информацию, необходимую для дублирования структуры, расположения и содержания данных какого-либо устройства хранения информации. Обычно образ диска просто повторяет набор секторов носителя, игнорируя файловую систему, построенную на нём.

Когда мы вставляем в компьютер диск или флешку, на самом низком уровне система получает к нему доступ. Она может обратиться к его битам и байтам.

Не факт, что операционная система сможет дать пользователю доступ к файлам и папкам на этом диске: он может быть зашифрован, файлы могут быть битыми или на диске может применяться неведомые операционке файловые системы. Но прочитать байты система всё равно может.

Если теперь прочитать все байты по очереди и в таком же порядке их записать в файл, мы получим точный образ диска. Весь мусор, всё шифрование, все битые данные — всё побайтно перенесётся в образ как точный физический слепок.

Теперь этот файл можно «примонтировать» к операционной системе — этой или какой-нибудь другой. И если операционка умеет работать с содержимым образа, она даст к этому доступ.

А если этот файл «прожечь» на флешку или другой диск, то все исходные байты окажутся на новом диске в той же последовательности. И если, например, исходный диск был загрузочным, то и новый «прожжённый» диск тоже станет загрузочным.

Знать, что такое образ диска и уметь пользоваться виртуальными приводами должен, на мой взгляд, любой, у кого дома есть компьютер. Образы дисков могут быть очень полезны не только любителям посмотреть кино и геймерам, но и обычным людям, стремящимся сохранить свои данные.

В первую очередь, образы дисков нужны для резервного копирования. Эта технология позволяет сделать копию любого носителя, с возможностью воспроизвести его в первозданном виде, даже если оригинал утерян.

Например, удобно иметь образ системного раздела компьютера или ноутбука с уже установленным программным обеспечением и произведенными настройками? Удобно, конечно! Если произойдет крах системы, то нам достаточно будет восстановить системный раздел из такого образа, сохраненного где-то в безопасном месте. Разница во времени, по сравнению с установкой системы и всего софта колоссальная. Большинство

разумных системных администраторов в компаниях, имеющих более десятка компьютеров так и постуупают.

Другой пример – диск с драйверами или с программным обеспечением для какого-то устройства. Такие диски потерять – легкое дело. А если заранее собрать образы таких носителей в одном месте, то, даже если не нашел оригинальные диски, можно не беспокоиться.

Кроме резервного копирования, образы дисков часто используются геймерами для того, чтобы ускорить работу с информацией на внешнем носителе. Доступ к образу диска, лежащему на HDD, происходит на порядок быстрее, чем доступ к CD или DVD в обычном дисководе. За счет этого часто получается значимый выигрыш в скорости загрузки игры и ее работы.

В виде образов дисков часто распространяется в сети интернет разнообразное ПО. Разумеется, очень много среди этих дисков нелегальных копий различных игр, операционных систем и т.п.

Нужно отметить, что ряд программ распространяется совершенно легально именно в виде образов дисков. Обычно, это всякого рода LiveCD – диски, с которых можно загрузить компьютер. Такие варианты имеются у всех ведущих антивирусов и большинства программ для работы с жесткими дисками. Так что далеко не все образы дисков в интернете пиратские.

Inode — это структура данных, в которой хранится информация о файле или директории в файловой системе. В файловой системе Linux, например ext4, у файла есть не только само его содержимое, например, текст, но и метаданные, такие как имя, дата создания, доступа, модификации и права. Эти метаданные и хранятся в inode. У каждого файла есть своя уникальная inode и именно здесь указано в каких блоках находятся данные файла.

Inode хранит метаданные для каждого файла в вашей системе в виде таблицы, обычно расположенной в начале раздела. Они хранят всю информацию, кроме имени файла и данных.

Каждый файл в данном каталоге является записью с именем файла и номером индекса. Вся остальная информация о файле извлекается из таблицы индексов путем ссылки на номер индекса.

Номера inodes уникальны на уровне раздела. Каждый раздел как собственная таблица индексов.

Если у вас закончились inode, вы не можете создавать новые файлы, даже если у вас есть свободное место на данном разделе.

Node означает индексный узел. Хотя история не совсем уверена в этом, это самое логичное и лучшее предположение, которое они придумали. Раньше было написан I-node, но дефис со временем потерял вес.

Inodes хранит метаданные о файле, к которому он относится. Эти метаданные содержат всю информацию об указанном файле.

- Размер.
- Разрешение.
- Владелец/группа.
- Расположение жесткого диска.

- Дата/время.
- Любая другая необходимая информация.

Каждый используемый inode ссылается на 1 файл. Каждый файл имеет 1 индекс. Каталоги, файлы символов, блочные устройства, все это файлы. У каждого из них есть 1 индекс.

Для каждого файла в каталоге есть запись, содержащая имя файла и номер индекса, связанный с ним.

Inodes являются уникальными на уровне разделов. Вы можете иметь два файла с одинаковым номером inode, если они находятся в другом разделе. Информация inode хранится в виде таблицы в виде структуры в стратегических частях каждого раздела. Часто встречается в начале.

Количество inode каждой файловой системы определяется при создании файловой системы. Для большинства пользователей число inode по умолчанию более чем достаточно.

Большинство настроек по умолчанию при создании файловой системы создает 1 inode на каждые 2 Кбайт пространства. Это дает множество inode для большинства систем. Скорее всего, вам не хватит места, прежде чем закончатся inode. При необходимости вы можете указать, сколько inode создавать при создании файловой системы.

Если у вас закончились inode, вы не сможете создать новый файл. Ваша система также не сможет это сделать. Это не та ситуация, с которой сталкивается большинство пользователей, но это возможно.

Например, почтовый сервер будет хранить огромное количество очень маленьких файлов. Многие из этих файлов будут меньше 2K байтов. Также ожидается постоянный рост. Поэтому почтовому серверу грозит нехватка inode, прежде чем закончится свободное место.

В некоторых файловых системах в Linux, таких как Btrfs, JFS, XFS, реализованы динамические inode. Они могут увеличить количество доступных inode, если это необходимо.

Когда создается новый файл, ему присваивается номер inode и имя файла. Номер индекса – это уникальный номер в этой файловой системе. И имя, и номер индекса хранятся в виде записи в каталоге.

Работа inode также объясняет, почему невозможно создать жесткую ссылку на другую файловую систему. Разрешение такой задачи откроет возможность наличия конфликтующих номеров inode. Мягкая ссылка, с другой стороны, может быть создана в другой файловой системе.

Поскольку жесткая ссылка имеет тот же номер inode, что и исходный файл, вы можете удалить исходный файл, и данные по-прежнему доступны через жесткую ссылку. Все, что вы сделали в этом случае, это удалите одно из имен, указывающих на этот номер inode. Данные, связанные с этим номером inode, будут оставаться доступными до тех пор, пока не будут удалены все связанные с ним имена.

Inode также являются важной причиной, по которой система Linux может обновляться без перезагрузки. Это связано с тем, что один процесс

может использовать файл библиотеки, в то время как другой процесс заменяет этот файл новой версией. Поэтому создаем новый индекс для нового файла. Уже запущенный процесс будет продолжать использовать старый файл, в то время как каждый новый вызов приведет к использованию новой версии.

Еще одна интересная особенность, которая поставляется с inode – это возможность хранить данные в самом inode. Это называется Inlining. Преимущество этого метода хранения заключается в экономии места, поскольку блок данных не потребуется. Это также увеличивает время поиска, избегая большего доступа к диску для получения данных.

В некоторых файловых системах, таких как ext4, есть опция inline_data. Когда он включен, он позволяет операционной системе хранить данные таким образом. Из-за ограничения размера вставка работает только для очень маленьких файлов. Ext2 и более поздние версии часто сохраняют информацию о мягких ссылках таким образом. То есть если размер не более 60 байт.

Inode – это не то, с чем вы взаимодействуете напрямую, но они играют важную роль. Если раздел должен содержать много очень маленьких файлов, таких как почтовый сервер, знание того, что они из себя представляют и как они работают, может спасти вас от многих проблем в будущем.

3 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Полностью ознакомившись с теоретическими аспектами реализуемого программного обеспечения, было решено структурно разделить приложение на четыре модуля, которые будут взаимодействовать друг с другом.

Первый модуль представляет собой метод получения косвенных блоков. Второй модуль – это блок чтения секторов. Третий модуль – блок получения записей каталога блока данных. Четвёртый блок – блок проверки inode.

3.1 Модуль получения указателей адресов блоков.

В этом модуле используется метод для чтения косвенного блока. Это позволяет нам читать номера блоков из косвенного блока.

3.2 Модуль чтения секторов.

В этом модуле непосредственно происходит чтение секторов из получаемого файла.

3.3 Модуль получения записей каталога блока данных.

В этом модуле происходит получение всех записей каталога для данного блока данных.

3.4 Модуль проверки inode

Данный модуль используется для проверки того, ссылается ли inode в каталоге или нет.

4 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

4.1 Описание используемых структур

4.1.1 Структура dinode

Поля:
- `short type` – тип файла;
- `short major` – основной номер устройства;
- `short minor` – младший номер устройства;
- `short nlink` – количество ссылок на inode в файловой системе;
- `uint size` – размер файла (биты);
- `uint addrs` – адреса блоков данных.

Данная структура выполняет роль указателя на inode на диске

4.1.2 Структура superblock

Поля:
- `uint nblocks` – количество блоков данных;
- `uint ninodes` – количество inode.

Данная структура указывает на суперблок файловой системы.

4.1.3 Структура dirent

Поля:
- `ushort inum` – количество inode;
- `char name` – имя каталога.

Данная структура описывает каталоги.

4.2 Описание используемых функций

- `uint xint` – используется для получения указателей адресов блоков. Взято из реализации mkfs.reference_count;
- `void rsect` – используется для чтения сектора из данного файла;
- `void get_dir_entries` – используется для получения всех записей каталога для данного блока данных;
- `void is_inode_in_dir` – используется для проверки того, ссылается ли данный inode в каталоге или нет;
- `void check_rule_1` – проверяет наличие неисправного inode. Если

размер меньше 0, значит индекс неисправен;

- `void check_rule_2_direct` – эта функция выполняет проверку действительности адреса, на который указывает блок данных;
- `void check_rule_2_indirect` – эта функция проверяет, указывает ли косвенный блок на допустимый блок данных. Иначе выводит ошибку;
- `void check_rule_3_for_size` – в данной функции происходит проверка на существование корневого каталога;
- `void check_rule_3_for_root_inode` – проверяет корректность корневого и текущего каталога.
- `int check_rule_4_for_present_dir_link` – функция проверяет правильность форматирования диска.
- `void check_rule_4_for_dir_type_and_format` – если inode не является каталогом или отсутствует в одном из файлов, то функция выводит сообщение о некорректном форматировании.
- `void check_rule_5` – проверяет, действителен ли адрес, при том, что блок данных не используется;
- `void check_rule_6` – если блок помечен как используемый, функция проверяет, действительно ли адрес используется или нет;
- `void check_rule_7` – функция указывает на ошибку, если адрес использовался более одного раза;
- `void check_rule_8` – выдает ошибку, если адрес блока уже используется;
- `void check_rule_9` – если inode помечен как используемый, но он не найден в каталоге, то функция выведет ошибку;
- `void check_rule_10` – функция проверяет на недопустимость ссылки на свободный тип inode;
- `void check_rule_11` – обрабатывает случай, если inode является файлом.
- `void check_rule_12` – функция выдаст ошибку, когда inode является каталогом, на который ссылаются более одного раза;
- `int main` - это основная функция.

5 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

5.1 Алгоритм работы функции void is_inode_in_dir:

Шаг 1. Перебираем все inode в суперблоке.
Шаг 2. Получаем количество записей.
Шаг 3. Перебираем все блоки данных, используемые каталогом.
Шаг 4. Получаем доступ к блокам каталогов и получаем количество inode.
Шаг 5. Обновляем количество inode каталога.
Шаг 6. Обновляем счетчик для количества ссылок на inode.

5.2 Алгоритм работы функции void get_dir_entries:

Шаг 1. Создаём структуру dirent.
Шаг 2. Инициализируем iterator.
Шаг 3. Проверяем вход в директорию.
Шаг 4. Обновляем количество inode.

6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ И РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ

Ext4 широко используемая файловая система. Данный проект представляет собой утилиту проверки целостности данной файловой системы. Для использования данной утилиты требуется её скомпилировать используя команду make.

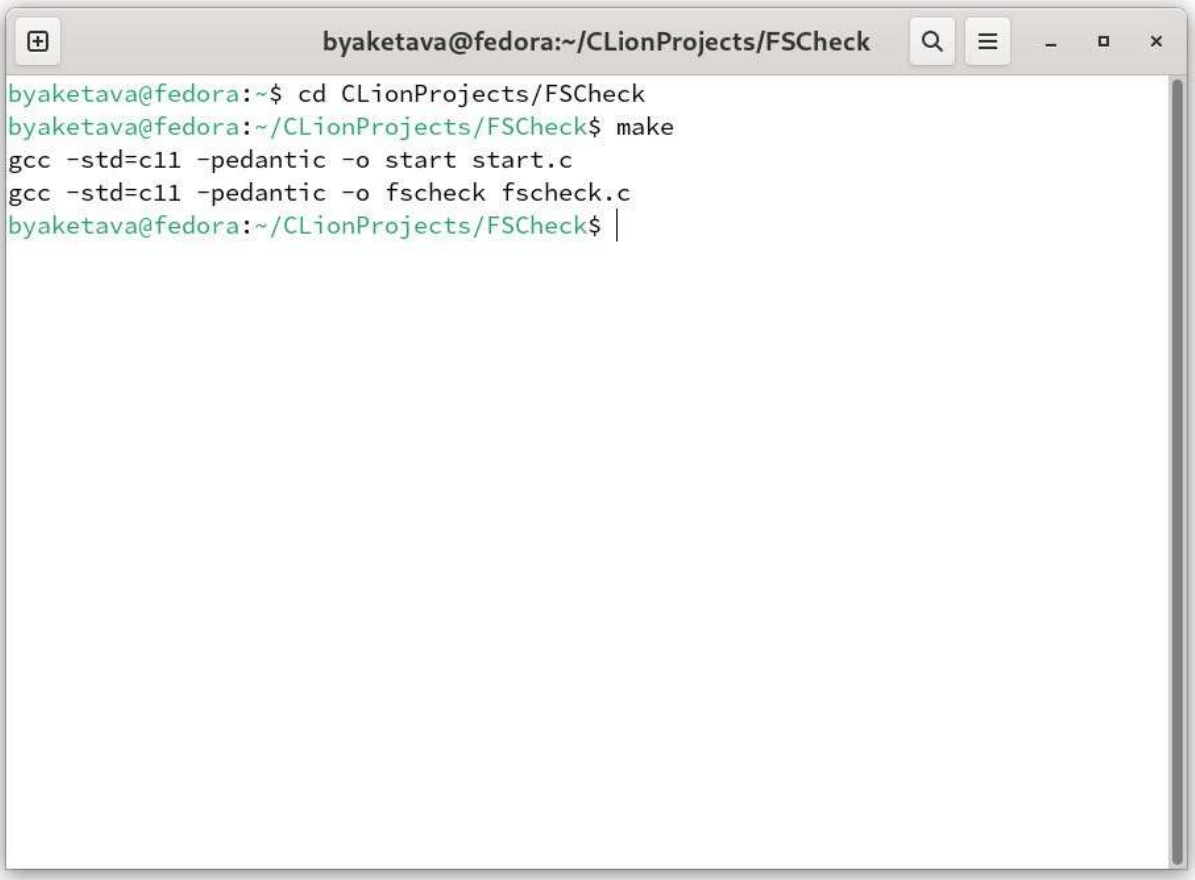

Рисунок 6.1 – Утилита fscheck

В качестве аргумента данная утилита принимает образ диска файловой системы. Так что требуется либо иметь существующий образ, либо создать новый.

Создать образ диска файловой системы можно следующим образом:

- С помощью команды `dd if=/dev/zero of=fs.img bs=1M count=256` создаём образ диска с именем fs.img размером 256 мб.
- С помощью команды `mkfs.ext4 fs.img` преобразуем диск в файловую систему ext4.

Далее можно монтировать образ диска файловой системы и заполнить данными для тестирования утилиты.

Перед запуском утилиты можно проверить состояние образа файловой системы с помощью программы start.

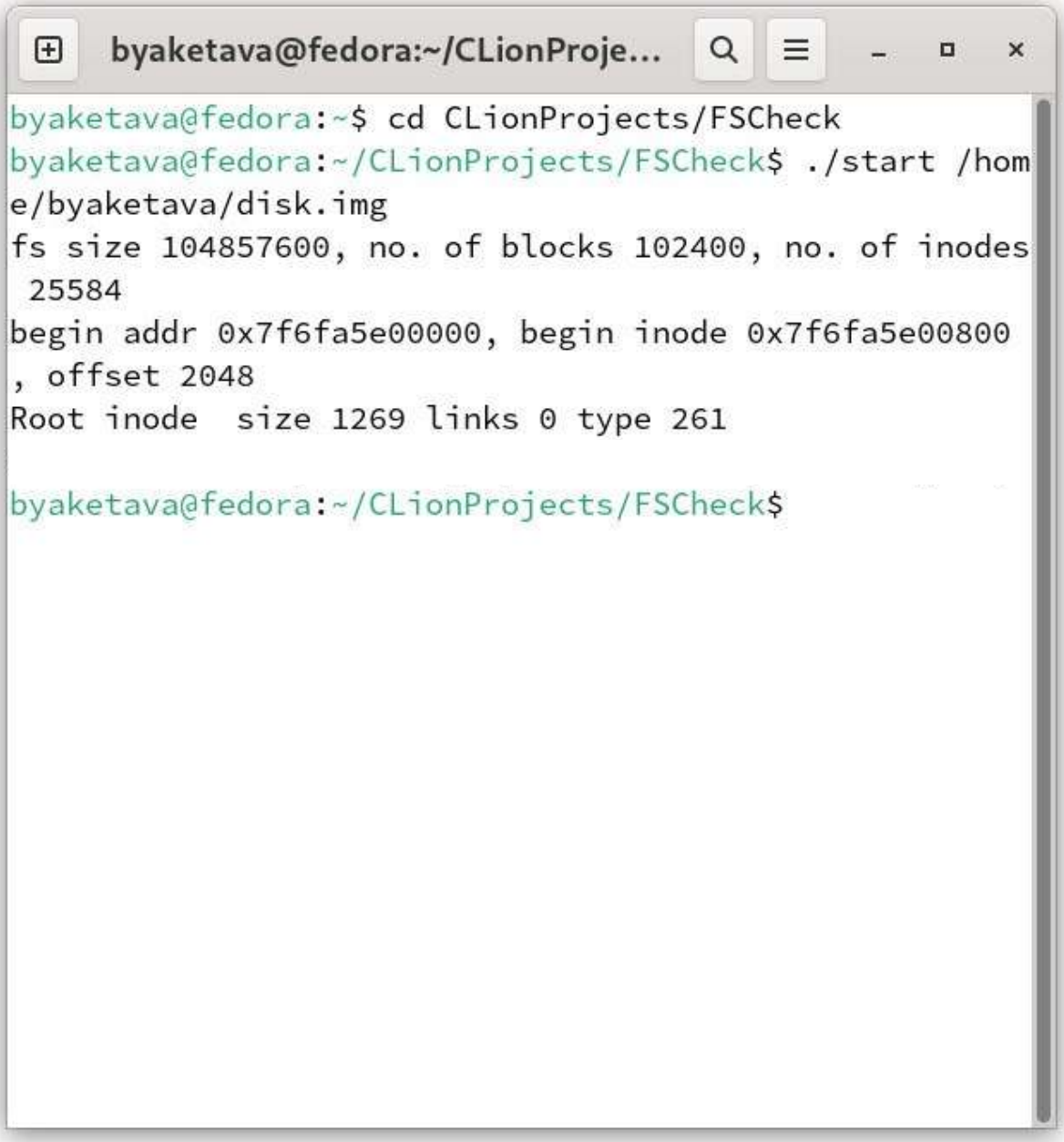

Рисунок 6.2 – Программа start

Теперь достаточно написать название утилиты и передать ей образ файловой системы.

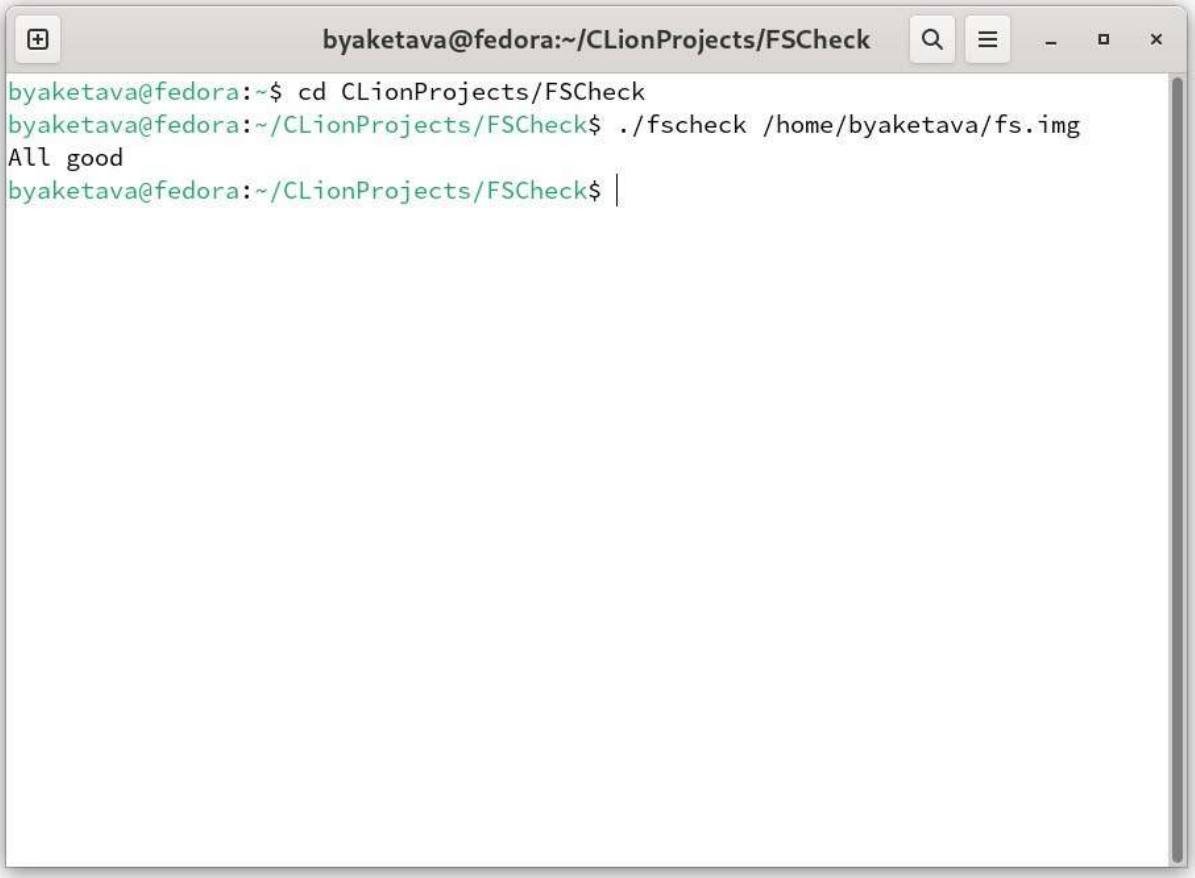

Рисунок 6.3 – Утилита fscheck

В случае, если образ содержит ошибки, то утилита выведет соответствующее сообщение.

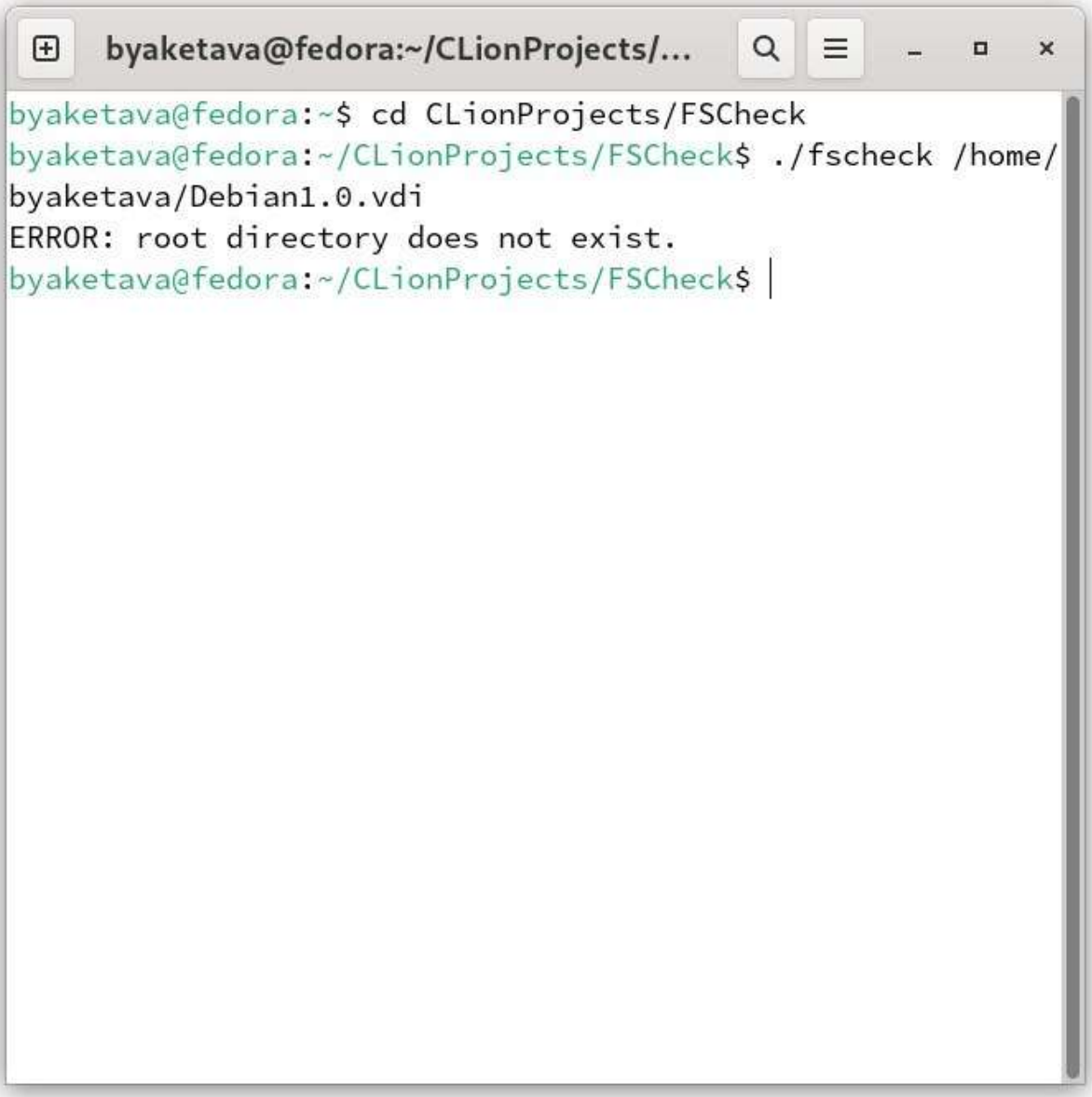

Рисунок 6.4 – Утилита fscheck

ЗАКЛЮЧЕНИЕ

Файловая система – это понятие, с которым столкнётся каждый пользователь Linux. В итоге была разработана утилита для работы с самой распространённой файловой системой ext4. Данный курсовой проект является результатом процесса, включающего анализ поставленной задачи, исследование существующих аналогов, работу с литературой, проектирование структур данных, разработку алгоритмов, тестирование созданной программы, составление руководства пользователя. Разработка данного приложения требовала познаний в области ОС Linux и её файловых систем.

К достоинствам программы можно отнести простой и понятный интерфейс, что в свою очередь обеспечивает удобство эксплуатации для обычных пользователей.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Лав, Р. Linux. Системное программирование: [изучаем ядро Linux и библиотеки C] / Роберт Лав; [перевод с английского О. Сивченко]. - Санкт-Петербург [и др.] : Питер : Питер Пресс, 2016. - 445 с.

[2] Таненбаум, Э.С. Современные операционные системы / Э. Таненбаум, Х. Бос; [перевели с английского А. Леонтьева, М. Малышева, Н. Вильчинский]. - Санкт-Петербург [и др.] : Питер : Питер Пресс, 2017. - 1119 с. - (Классика computer science).

[3] Мэтью, Н. Основы программирования в Linux : [перевод с английского] / Нейл Мэтью, Ричард Стоунс. - Санкт-Петербург : БХВ-Петербург, 2009. - XIV, 882 с.

[4] 1. Структура и типы файловых систем в Linux [Электронный ресурс]. Электронные данные. -Режим доступа: https://selectel.ru/blog/directory-structure-linux/ - Дата доступа: 02.05.2024.

[5] Живицкая, Е.Н. Положение о организации и проведении курсового проектирования в БГУИР 03–0018–2013. / Е.Н. Живицкая [и др.]– Минск: БГУИР, 2013 – 17с.

ПРИЛОЖЕНИЕ А
(обязательное)
Схема структурная

ПРИЛОЖЕНИЕ Б
(обязательное)
Диаграмма последовательности

ПРИЛОЖЕНИЕ В
(обязательное)
Ведомость документов

ПРИЛОЖЕНИЕ Г
(обязательное)
Листинг кода

Файл fscheck.c:

```
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "types.h"
#include "fs.h"

#define DIR_ENTRY_PER_BLOCK (BSIZE / sizeof(struct dirent))
#define T_DIR   1   //For determining type of inode - Directory
#define T_FILE  2   //For determining type of inode - File
#define T_DEV   3   //For determining type of inode - Device

struct dinode *disk_inodes_arr = NULL;
struct superblock *super_block = NULL;
int *referenced_inodes_arr = NULL;

uint xint(uint x)
{
    uint indirect_block_addr = 0;
    char *a = (char *) &indirect_block_addr;
    a[0] = x;
    a[1] = x >> 8;
    a[2] = x >> 16;
    a[3] = x >> 24;
    return indirect_block_addr;
}

void rsect(int image_file_handler, uint sec, void *buf)
{
    if (lseek(image_file_handler, sec * 1024L, 0) != sec * 1024L)
    {
        perror("lseek");
        exit(1);
    }

    if (read(image_file_handler, buf, 1024) != 1024)
    {
        perror("read");
        exit(1);
    }
}

void get_dir_entries(uint block_number, char *mmap_address_space,
uint *num_inodes)
```

```
{
    struct dirent *directory_entry = (struct dirent *)
(mmap_address_space + block_number * BSIZE);
    int iterator = 0;
    for (iterator = 0; iterator < DIR_ENTRY_PER_BLOCK; iterator++,
directory_entry++)
    {
        if ((strcmp(directory_entry->name, ".") == 0) ||
(strcmp(directory_entry->name, "..") == 0))
        {
            num_inodes[iterator] = 0;
            continue;
        }
        num_inodes[iterator] = directory_entry->inum;
    }
}

void is_inode_in_dir(int image_file_handler, char
*mmap_address_space)
{
    int outer_iterator = 0;
    int inner_iterator = 0;

    for (outer_iterator = 0; outer_iterator < super_block->ninodes;
outer_iterator++)
    {
        //If inode represents a directory.
        if (disk_inodes_arr[outer_iterator].type == T_DIR)
        {
            //Get number of entries
            int num_dir_entries =
disk_inodes_arr[outer_iterator].size / sizeof(struct dirent);
            for (inner_iterator = 0; inner_iterator < NDIRECT;
inner_iterator++)
            {
                if
((disk_inodes_arr[outer_iterator].addrs[inner_iterator] == 0) ||
(num_dir_entries <= 0))
                {
                    continue;
                }
                uint *num_inodes = (uint *)
calloc(DIR_ENTRY_PER_BLOCK, sizeof(uint));
                if (num_inodes == NULL)
                {
                    perror("Memory allocation failed");
                    exit(1);
                }

get_dir_entries(disk_inodes_arr[outer_iterator].addrs[inner_iterato
r], mmap_address_space, num_inodes);
```

```
                int temp_iterator = 0;
                for (temp_iterator = 0; temp_iterator <
DIR_ENTRY_PER_BLOCK; temp_iterator++)
                {
                    if (num_inodes[temp_iterator] != 0)
                    {

referenced_inodes_arr[num_inodes[temp_iterator]]++;
                        num_dir_entries--;
                    }
                }
                free(num_inodes);
            }
            if (disk_inodes_arr[outer_iterator].addrs[NDIRECT] ==
0)
            {
                continue;
            }
            uint indirect[NINDIRECT] = {0};
            rsect(image_file_handler,
xint(disk_inodes_arr[outer_iterator].addrs[NDIRECT]), (char *)
indirect);
            for (inner_iterator = 0; inner_iterator < NINDIRECT;
inner_iterator++)
            {
                if ((indirect[inner_iterator] == 0) ||
(num_dir_entries <= 0))
                {
                    continue;
                }
                uint *num_inodes = (uint *)
calloc(DIR_ENTRY_PER_BLOCK, sizeof(uint));
                if (num_inodes == NULL)
                {
                    perror("Memory allocation failed");
                    exit(1);
                }
                get_dir_entries(indirect[inner_iterator],
mmap_address_space, num_inodes);


                int temp_iterator = 0;
                for (temp_iterator = 0; temp_iterator <
DIR_ENTRY_PER_BLOCK; temp_iterator++)
                {
                    if (num_inodes[temp_iterator])
                    {

referenced_inodes_arr[num_inodes[temp_iterator]]++;
                        num_dir_entries--;
                    }
                }
```

```
                free(num_inodes);
            }
        }
    }
}

void check_rule_1(struct dinode *disk_inodes_arr, int iterator)
{
    if ((disk_inodes_arr[iterator].size < 0) ||
(disk_inodes_arr[iterator].type < T_DIR) ||
        (disk_inodes_arr[iterator].type > T_DEV))
    {
        fprintf(stderr, "ERROR: bad inode.\n");
        exit(1);
    }
}

void check_rule_2_direct(struct dinode *disk_inodes_arr, int
outer_iterator, int inner_iterator)
{
    if ((disk_inodes_arr[outer_iterator].addrs[inner_iterator] < 0)
||
        (disk_inodes_arr[outer_iterator].addrs[inner_iterator] >
super_block->nblocks))
    {
        fprintf(stderr, "ERROR: bad direct address in inode.\n");
        exit(1);
    }
}

void check_rule_2_indirect(int indirect_block_num)
{
    if ((indirect_block_num < 0) || (indirect_block_num >
super_block->nblocks))
    {
        fprintf(stderr, "ERROR: bad indirect address in inode.\n");
        exit(1);
    }
}

void check_rule_3_for_size(int size)
{
    if (size <= 0)
    {
        fprintf(stderr, "ERROR: root directory does not exist.\n");
        exit(1);
    }
}

void check_rule_3_for_root_inode(int inode_num)
{
    if (inode_num != ROOTINO)
```

```
    {
        fprintf(stderr, "ERROR: root directory does not exist.\n");
        exit(1);
    }
}

int check_rule_4_for_present_dir_link(struct dirent
*directory_entry,
                                      int outer_iterator, int
reference_count)
{
    if (directory_entry->inum == outer_iterator)
    {
        reference_count++;
    } else
    {
        fprintf(stderr, "ERROR: directory not properly
formatted.\n");
        exit(1);
    }
    return reference_count;
}

void check_rule_4_for_dir_type_and_format(short type, int
reference_count)
{
    if ((type == T_DIR) && (reference_count != 2))
    {
        fprintf(stderr, "ERROR: directory not properly
formatted.\n");
        exit(1);
    }
}

void check_rule_5(char *block_usage_bitmap, uint block_num)
{
    if (((block_usage_bitmap[block_num / 8]) & (0x1 << (block_num %
8))) == 0)
    {
        fprintf(stderr, "ERROR: address used by inode but marked
free in bitmap.\n");
        exit(1);
    }
}

void check_rule_6(char *mmap_address_space, uint first_block, uint
*used_blocks_arr)
{
    int interator = 0;
    for (interator = 0; interator < first_block + super_block-
>nblocks; interator++)
    {
```

```
        char *block_usage_bitmap = mmap_address_space +
(BBLOCK(interator, super_block->ninodes)) * BSIZE;
        if (((block_usage_bitmap[interator / 8]) & (0x1 <<
(interator % 8))))
        {
            if (used_blocks_arr[interator] == 0)
            {
                fprintf(stderr, "ERROR: bitmap marks block in use
but it is not in use.\n");
                exit(1);
            }
        }
    }
}

void check_rule_7(uint is_used)
{
    if (is_used == 1)
    {
        fprintf(stderr, "ERROR: direct address used more than
once.\n");
        exit(1);
    }
}

void check_rule_8(uint is_used)
{
    if (is_used)
    {
        fprintf(stderr, "ERROR: indirect address used more than
once.\n");
        exit(1);
    }
}

void check_rule_9(int is_used)
{
    if (is_used == 0)
    {
        fprintf(stderr, "ERROR: inode marked use but not found in a
directory.\n");
        free(referenced_inodes_arr);
        exit(1);
    }
}

void check_rule_10(int is_used)
{
    if (is_used != 0)
    {
        fprintf(stderr, "ERROR: inode referred to in directory but
marked free.\n");
```

```
        free(referenced_inodes_arr);
        exit(1);
    }
}

void check_rule_11(short type, int ref_inode_num, short nliks)
{
    //If the inode is a file. It should be referred for a total
number of times same as its number of links. Else throw error.
Rule-11: Referrence count is bad.
    if ((type == T_FILE) && (ref_inode_num != nliks))
    {
        fprintf(stderr, "ERROR: bad reference count for file.\n");
        free(referenced_inodes_arr);
        exit(1);
    }
}

void check_rule_12(short type, int ref_inode_num)
{
    if ((type == T_DIR) && (ref_inode_num > 1))
    {
        fprintf(stderr, "ERROR: directory appears more than once in
file system.\n");
        free(referenced_inodes_arr);
        exit(1);
    }
}

int main(int argc, char *argv[])
{
    int image_file_handler = 0;
    char *mmap_address_space = NULL;
    struct dirent *directory_entry = NULL;
    struct stat file_statistics;

    if (argc != 2)
    {
        fprintf(stderr, "Usage: fscheck <file_system_image>\n");
        exit(1);
    }

    image_file_handler = open(argv[1], O_RDONLY);
    if (image_file_handler < 0)
    {
        fprintf(stderr, "Image not found.\n");
        exit(1);
    }

    fstat(image_file_handler, &file_statistics);

    mmap_address_space = mmap(NULL, file_statistics.st_size,
```

```
                                PROT_READ, MAP_PRIVATE,
image_file_handler, 0);
    if (mmap_address_space == MAP_FAILED)
    {
        perror("Mmap failed");
        exit(1);
    }

    super_block = (struct superblock *) (mmap_address_space +
BSIZE);
    uint fs_size = (super_block->nblocks * BSIZE);
    disk_inodes_arr = (struct dinode *) (mmap_address_space +
IBLOCK((uint) 0) * BSIZE);

    int num_inodes = super_block->ninodes;
    int outer_iterator, inner_iterator;

    check_rule_3_for_size(disk_inodes_arr[ROOTINO].size);

    directory_entry = (struct dirent *) (mmap_address_space +
(disk_inodes_arr[ROOTINO].addrs[0]) * BSIZE);
    int size = disk_inodes_arr[ROOTINO].size / sizeof(struct
dirent);
    int reference_count = 0;

    for (outer_iterator = 0; outer_iterator < size;
outer_iterator++)
    {
        if ((reference_count < 2) && ((strcmp(directory_entry-
>name, ".") == 0)
                                      || (strcmp(directory_entry-
>name, "..") == 0)))
        {
            check_rule_3_for_root_inode(directory_entry->inum);
            reference_count++;
        }
        directory_entry++;
    }

    uint used_blocks_arr[fs_size];
    uint first_block = BBLOCK(fs_size, super_block->ninodes) + 1;

    for (outer_iterator = 0; outer_iterator < fs_size;
outer_iterator++)
    {
        used_blocks_arr[outer_iterator] = 0;
    }
    for (outer_iterator = 0; outer_iterator < first_block;
outer_iterator++)
    {
        used_blocks_arr[outer_iterator] = 1;
    }
```

```
    for (outer_iterator = 0; outer_iterator < num_inodes;
outer_iterator++)
    {
        if (disk_inodes_arr[outer_iterator].size == 0)
        {
            continue;
        }
        check_rule_1(disk_inodes_arr, outer_iterator);
        reference_count = 0;
        for (inner_iterator = 0; inner_iterator < NDIRECT;
inner_iterator++)
        {
            if
(disk_inodes_arr[outer_iterator].addrs[inner_iterator] == 0)
            {
                continue;
            }
            check_rule_2_direct(disk_inodes_arr, outer_iterator,
inner_iterator);
            uint block_num =
disk_inodes_arr[outer_iterator].addrs[inner_iterator];
            char *block_usage_bitmap = mmap_address_space +
(BBLOCK(block_num, super_block->ninodes)) * BSIZE;

            check_rule_5(block_usage_bitmap, block_num);
            check_rule_7(used_blocks_arr[block_num]);
            used_blocks_arr[block_num] = 1;
            if (disk_inodes_arr[outer_iterator].type == T_DIR)
            {
                directory_entry = (struct dirent *)
(mmap_address_space + block_num * BSIZE);
                int size = disk_inodes_arr[outer_iterator].size /
sizeof(struct dirent);


                int temp_iterator = 0;
                for (temp_iterator = 0; temp_iterator < size;
temp_iterator++, directory_entry++)
                {
                    if (reference_count < 2)
                    {
                        if (strcmp(directory_entry->name, ".") ==
0)
                        {
                            reference_count =
check_rule_4_for_present_dir_link(directory_entry, outer_iterator,

reference_count);
                        } else if (strcmp(directory_entry->name,
"..") == 0)
                        {
                            reference_count++;
```

```
                        }
                    }
                }
            }
        }

check_rule_4_for_dir_type_and_format(disk_inodes_arr[outer_iterator
].type, reference_count);

        if (disk_inodes_arr[outer_iterator].addrs[NDIRECT] == 0)
        {
            continue;
        }

        uint temp_inode_addr =
disk_inodes_arr[outer_iterator].addrs[NDIRECT];
        if ((temp_inode_addr > 0) && (temp_inode_addr < fs_size))
        {
            used_blocks_arr[temp_inode_addr] = 1;
        }

        uint indirect[NINDIRECT] = {0};
        uint indirect_block_addr = xint(temp_inode_addr);
        rsect(image_file_handler, indirect_block_addr, (char *)
indirect);

        for (inner_iterator = 0; inner_iterator < NINDIRECT;
inner_iterator++)
        {
            uint indirect_block_num = indirect[inner_iterator];
            if (indirect_block_num == 0)
            {
                continue;
            }

            check_rule_2_indirect(indirect_block_num);
            char *block_usage_bitmap = mmap_address_space +
(BBLOCK(indirect_block_num, super_block->ninodes)) * BSIZE;
            check_rule_5(block_usage_bitmap, indirect_block_num);
            check_rule_8(used_blocks_arr[indirect_block_num]);
            used_blocks_arr[indirect_block_num] = 1;
        }
    }

    check_rule_6(mmap_address_space, first_block, used_blocks_arr);
    referenced_inodes_arr = (int *) calloc(super_block->ninodes,
sizeof(int));
    if (referenced_inodes_arr == NULL)
    {
        perror("Memory allocation failed");
        exit(1);
    }
```

```
    is_inode_in_dir(image_file_handler, mmap_address_space);
    for (outer_iterator = 2; outer_iterator < super_block->ninodes;
outer_iterator++)
    {
        if (disk_inodes_arr[outer_iterator].type == 0)
        {
            check_rule_10(referenced_inodes_arr[outer_iterator]);
            continue;
        }
        check_rule_9(referenced_inodes_arr[outer_iterator]);
        check_rule_11(disk_inodes_arr[outer_iterator].type,
                      referenced_inodes_arr[outer_iterator],
                      disk_inodes_arr[outer_iterator].nlink);
        check_rule_12(disk_inodes_arr[outer_iterator].type,
                      referenced_inodes_arr[outer_iterator]);
    }
    printf("All good\n");
    exit(0);
}
```

Файл start.c:

```
#include <stdio.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "types.h"
#include "fs.h"

int main(int argc, char *argv[])
{
    int i, n, fsfd;
    char *addr;
    struct dinode *dip;
    struct superblock *sb;
    struct dirent *de;

    if (argc < 2)
    {
        fprintf(stderr, "Usage: sample fs.img ...\n");
        exit(1);
    }

    fsfd = open(argv[1], O_RDONLY);
    if (fsfd < 0)
    {
        perror(argv[1]);
        exit(1);
    }
```

```
    addr = mmap(NULL, BSIZE * 102400, PROT_READ, MAP_PRIVATE, fsfd,
0);
    if (addr == MAP_FAILED)
    {
        perror("mmap failed");
        exit(1);
    }

    sb = (struct superblock *) (addr + 1 * BSIZE);
    printf("fs size %d, no. of blocks %d, no. of inodes %d \n", sb-
>nblocks * BSIZE,
           sb->nblocks, sb->ninodes);

    dip = (struct dinode *) (addr + IBLOCK((uint) 0) * BSIZE);
    printf("begin addr %p, begin inode %p , offset %ld \n", addr,
           dip, (char *) dip - addr);


    printf("Root inode  size %d links %d type %d \n",
           dip[ROOTINO].size, dip[ROOTINO].nlink,
dip[ROOTINO].type);

    de = (struct dirent *) (addr + (dip[ROOTINO].addrs[0]) *
BSIZE);

    n = dip[ROOTINO].size / sizeof(struct dirent);
    for (i = 0; i < n; i++, de++)
    {
        printf(" inum %d, name %s ", de->inum, de->name);
        printf("inode  size %d links %d type %d \n",
               dip[de->inum].size, dip[de->inum].nlink, dip[de-
>inum].type);
    }
    exit(0);
}
```

Файл types.h

```
#pragma once

typedef unsigned int uint;
typedef unsigned short ushort;
```

Файл fs.h

```
#pragma once

#define ROOTINO 2
#define BSIZE 1024

struct superblock
{
    uint ninodes;
    uint nblocks;
```

```
};

#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)

struct dinode
{
    short type;
    short major;
    short minor;
    short nlink;
    uint size;
    uint addrs[NDIRECT + 1];
};

#define IPB (BSIZE / sizeof(struct dinode))
#define IBLOCK(i) ((i) / IPB + 2)
#define BPB (BSIZE*8)
#define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
#define DIRSIZ 14

struct dirent
{
    ushort inum;
    char name[DIRSIZ];
};
```