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

К ЗАЩИТЕ ДОПУСТИТЬ

_____ Б. В. Никульшин

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

«Утилита контроля появления дубликатов в файловой системе с заменой их на жесткие ссылки и протоколирования фактов замены»

БГУИР КП 1-40 02 01 003 ПЗ

Студент: Ермоленко А. А

Руководитель: старший преподаватель кафедры ЭВМ
Поденок Л. П.

Минск 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные файловые системы нередко сталкиваются с проблемой дублирования файлов, что приводит к избыточному использованию дискового пространства и снижению эффективности работы системы. Утилиты для поиска и устранения дубликатов файлов играют важную роль в оптимизации файловых хранилищ.

Одним из решений этой проблемы является замена дубликатов на жёсткие ссылки — специальный тип ссылок, позволяющий разным файлам указывать на одно и то же содержимое на диске без его дублирования. Это не только освобождает значительное количество дискового пространства, но и способствует улучшению производительности системы за счет снижения нагрузки на файловую систему при работе с дублированными данными. Такой подход также минимизирует риски ошибок, связанных с ручным управлением файлами и позволяет поддерживать целостность данных при частых изменениях файлов в системе.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Обзор предметной области

Проблема дублирования данных в файловых системах актуальна для множества современных пользователей и организаций. С ростом объемов информации, хранящейся на серверах и локальных устройствах, возникают ситуации, когда одно и то же содержимое может быть сохранено в разных местах, что приводит к ненужному расходованию дискового пространства и ухудшению производительности системы. В условиях ограниченности ресурсов, таких как хранилища и вычислительная мощность, эффективное управление файловыми данными становится приоритетной задачей.

Одним из ключевых механизмов борьбы с дубликатами является создание жестких ссылок. Жесткая ссылка позволяет нескольким файлам указывать на одно и то же физическое место на диске, что позволяет уменьшить объем используемого пространства без потери доступа к данным. В отличие от символических ссылок, жесткие ссылки не зависят от местоположения оригинального файла и продолжают функционировать, даже если исходный файл перемещен или удален, до тех пор, пока остается хотя бы одна ссылка на его содержимое. Этот механизм широко используется в различных файловых системах, таких как ext4 (Linux), NTFS (Windows), и помогает повысить эффективность использования хранилищ.

Протоколирование фактов замены дубликатов файлов на жесткие ссылки — важный аспект работы утилиты, который обеспечивает прозрачность операций, повышает безопасность и позволяет отслеживать изменения в файловой системе. Основные задачи протоколирования:

– запись информации о дубликатах и их замене;
– протоколирование ошибок и исключений;
– возможность восстановления исходного состояния;
– гибкость настройки журналов.

1.2 Анализ аналогов программного средства

В рамках разработки утилиты контроля появления дубликатов в файловой системе с заменой их на жесткие ссылки и протоколирования фактов замены важно изучить существующие программные средства, выполняющие аналогичные функции. Это позволяет не только понимать текущие технологии и подходы, но и выявить их сильные и слабые стороны, что поможет создать более эффективный и функциональный инструмент. В этом разделе рассмотрены наиболее известные и широко используемые аналоги.

1.2.1 fdupes

fdupes – это простая консольная утилита, которая находит дубликаты файлов на основе их содержимого. Она позволяет рекурсивно сканировать директории и взаимодействовать с пользователем для удаления дубликатов.

Программа имеет следующие возможности:

– поиск дубликатов файлов на основе сравнения хэш-сумм и последующего побайтового сравнения для точности;

– поддержка рекурсивного поиска по каталогам;

– поддержка удаления дубликатов и интерактивного выбора действий.

Недостатки приложения:

– отсутствие продвинутых настроек поиска, таких как фильтрация по типам файлов или времени последнего изменения;

– ограниченная функциональность в плане протоколирования операций.

1.2.2 rdfind

rdfind – CLI-инструмент, специализирующийся на поиске дубликатов и их удалении. Он создает текстовый отчет о найденных дубликатах и может быть настроен для автоматической замены дубликатов на жесткие ссылки. Rdfind эффективно управляет дубликатами, но ограничен в плане гибкости настроек, и у него отсутствует графический интерфейс.

Преимущества rdfind:

– поддержка более точного алгоритма хэширования (SHA-1);

– возможность гибкой настройки поиска с фильтрацией определенных каталогов и файлов;

– простота интеграции в автоматические скрипты и системы управления файлами.

Недостатки rdfind:

– ограниченная поддержка протоколирования операций — информация об изменениях сохраняется минимально;

– нет функции восстановления файлов после замены на жесткие ссылки.

1.2.3 Duplicate Cleaner

Duplicate Cleaner – утилита, ориентированная на поиск и управление дубликатами файлов с продвинутыми возможностями и графическим интерфейсом. Интерфейс и частичный функционал изображены на рисунках 1.2.3 и 1.2.4.

Возможности и преимущества утилиты:

– поддерживает поиск дубликатов по имени, размеру, содержимому и метаданным;

– подробный и настраиваемый интерфейс для управления результатами поиска;

– включает возможность предварительного просмотра файлов, настройки фильтров и отчеты;

– профессиональная версия позволяет заменять дубликаты на жесткие ссылки;

– детализированный поиск и управление результатами, возможность создания отчетов.

Ограничения программы:

– некоторые расширенные возможности доступны только в платной версии.

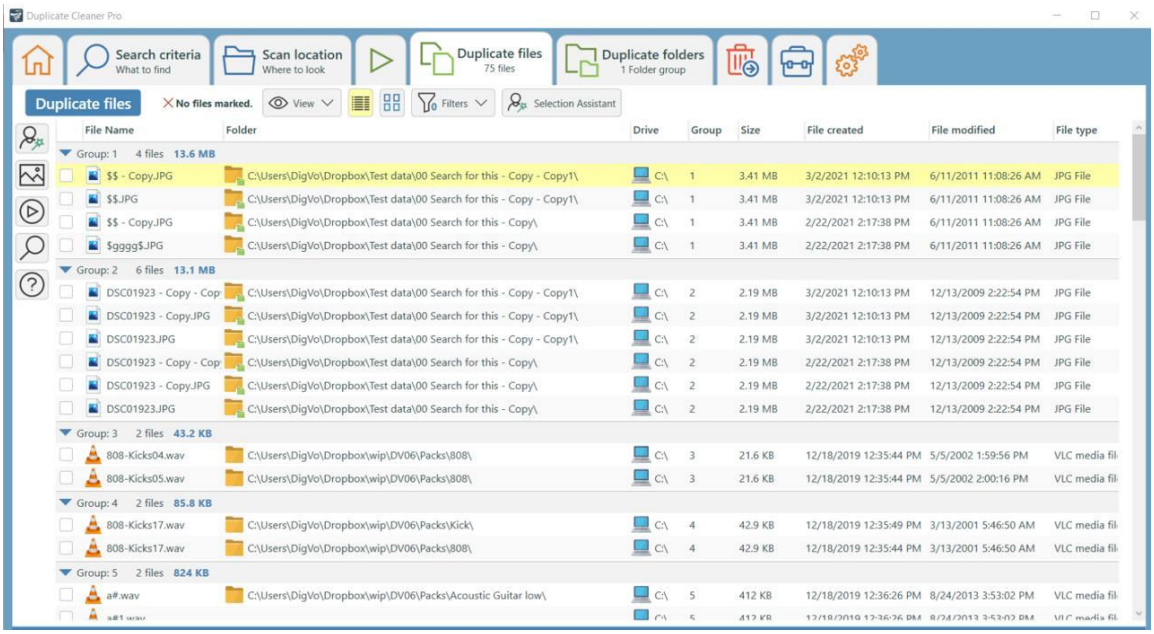

Рисунок 1.2.3 – Скриншот программы Duplicate Cleaner

1.2.4 Czkawka

Czkawka — это мощная и быстрая утилита с открытым исходным кодом для поиска дубликатов файлов и оптимизации дискового пространства. Она поддерживает Linux, Windows и macOS, и предоставляет как графический интерфейс (GUI), так и возможность использования через командную строку (CLI). Интерфейс и частичный функционал изображен на рисунке 1.2.4.

Основные функции Czkawka:

– поиск дубликатов файлов на основе хеширования файлов (SHA-1);

– помогает удалить ненужные каталоги, которые остались после удаления файлов;

– позволяет выявить файлы, занимающие больше всего места на диске;
– выявляет и удаляет битые символические ссылки.
Недостатки утилиты:
– не поддерживает выборочную сортировку результатов;
– ограниченная возможность предпросмотра.

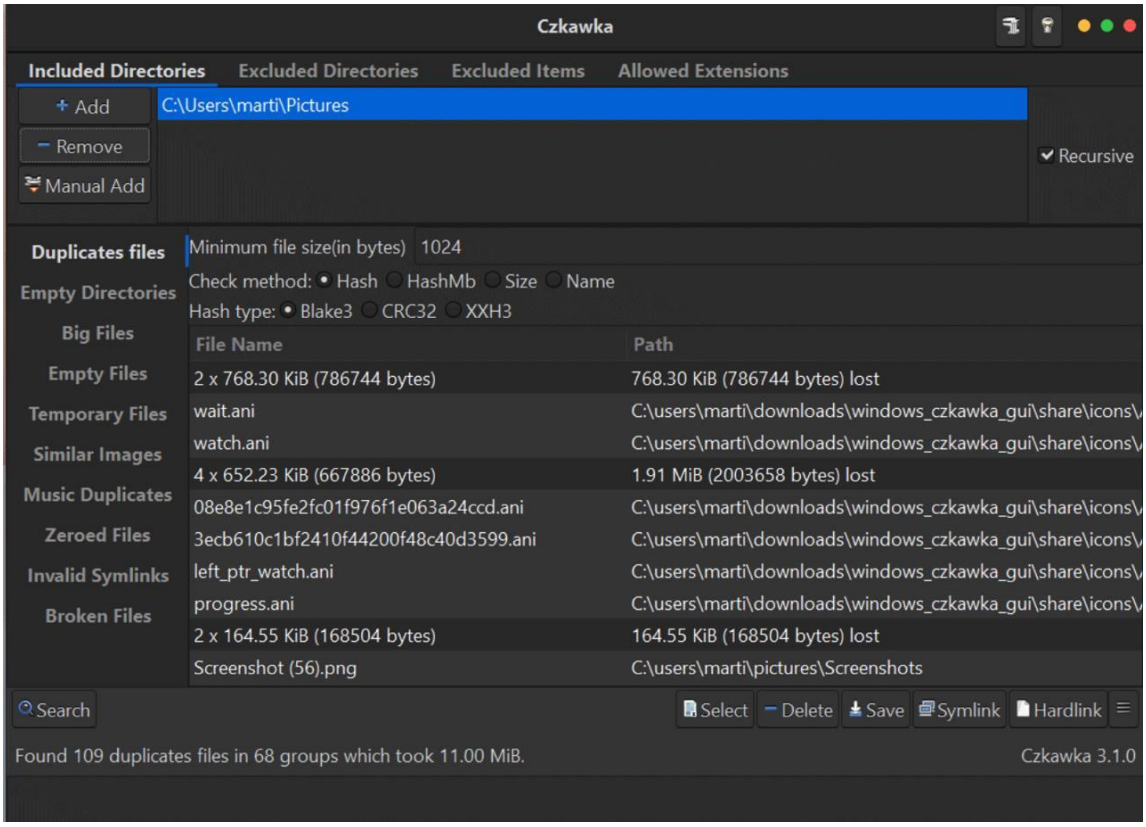

Рисунок 1.2.4 – Скриншот программы Czkawka

1.3 Постановка задачи

После рассмотрения аналогов можно сказать, что все они обладают большим количеством функций, которые невозможно реализовать в курсовом проекте за данный период времени. Поэтому были выбраны несколько ключевых возможностей, которые будут выполнены в рамках одного семестра.

Будет разработано приложение, удовлетворяющее следующим критериям:
– ввод директории пользователем;
– создание фонового процесса для проверки файлов;
– удаление фонового процесса;
– замена дублирующегося файла жёсткой ссылкой;
– логирование результатов фонового процесса;
– контроль за появлением дубликатов в файловой системе.

2 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описывается функционирование и структура разрабатываемого приложения.

2.1 Структура входных и выходных данных

Программа принимает в качестве входных данных командные аргументы:

– kill. Если этот аргумент указан при запуске, программа попытается завершить работающий процесс, который использует файл блокировки (LOCK_FILE).

– директория для сканирования. Если указан один аргумент (без -kill), он интерпретируется как путь к директории, которую нужно сканировать. Если аргументы не указаны, используется директория по умолчанию (DEFAULT_DIR).

В качестве выходных данных Лог-файл (LOG_FILE). Программа записывает сообщения о своем выполнении в лог-файл. Сообщения могут включать ошибки, информацию о завершении работы, успешные операции и другие события.

2.2 Описание работы функций

2.2.1 Функция main

Функция принимает параметры:

– `int argc` — количество аргументов командной строки;

– `char *argv[]` — массив строк, содержащий аргументы командной строки.

Возвращаемое значение: функция возвращает либо 0 при успешном завершении работы программы, либо EXIT_FAILURE в случае ошибки.

Функция запускает программу, которая в фоновом режиме периодически сканирует указанную директорию, записывая логи о начале и окончании каждого сканирования. Также она устанавливает обработчики для сигналов завершения и ошибки, создает файл блокировки и работает бесконечно, выполняя очистку и сканирование директории с заданным интервалом.

2.2.2 Функция log_message

Функция принимает параметры:

– `const char *type` — тип сообщения;

– `const char *message` — само сообщение для записи в лог.

Возвращаемое значение: функция log_message ничего не возвращает (тип `void`).

Функция записывает сообщение в лог-файл, добавляя к нему текущую метку времени и тип сообщения (например, "INFO" или "ERROR"). Если открыть лог-файл не удается, программа логирует ошибку в системный журнал через `syslog()`, удаляет временные файлы и завершает работу.

2.2.3 Функция check_same_inode

Функция принимает параметры:

– `const char *path1` — путь к первому файлу;

– `const char *path2` — путь ко второму файлу.

Возвращаемое значение: функция возвращает либо 1, если файлы имеют одинаковые inode и находятся на одном устройстве, либо 0, если файлы различаются.

Функция сравнивает два файла по их inode и устройству, чтобы проверить, являются ли они одинаковыми физическими файлами. Если не удается получить информацию о любом из файлов, она записывает ошибку в лог и завершает программу.

2.2.4 Функция write_file_info

Функция принимает параметры:

– `const FileInfo *file_info` — структура, содержащая информацию о файле (длина пути, путь, размер файла, хеш).

Возвращаемое значение: функция ничего не возвращает (тип `void`).

Функция записывает информацию о файле в бинарный файл, открывая его в режиме добавления. Она записывает длину пути, сам путь, размер файла и его хеш. Если открыть бинарный файл не удается, функция логирует ошибку и завершает программу.

2.2.5 Функция delete_info_by_path

Функция принимает параметры:

– `const char *search_path` — путь файла, информацию о котором нужно удалить.

Возвращаемое значение: функция ничего не возвращает (тип `void`).

Функция удаляет информацию о файле с указанным путем из бинарного файла. Она создает временный файл, копирует в него все записи, кроме записи с совпадающим путем, затем заменяет оригинальный файл

временным. Если запись с указанным путем найдена и удалена, логируется успешное удаление.

2.2.6 Функция find_duplicate

Функция принимает параметры:

– `const FileInfo *new_file_info` — информация о новом файле (длина пути, путь, размер, хеш).

Возвращаемое значение: функция возвращает путь к дубликату (тип `char*`), если дубликат найден. Возвращает NULL, если дубликатов нет.

Функция ищет дубликат файла, сравнивая размер и хеш нового файла с уже записанными файлами в бинарном файле. Если найден дубликат, функция возвращает путь к нему, иначе возвращает NULL.

2.2.7 Функция compute_md5

Функция принимает параметры:

– `const char *path` — путь к файлу, для которого нужно вычислить хеш;

– `char *hash_str` — строка, в которую будет записан вычисленный MD5-хеш в шестнадцатеричном виде.

Возвращаемое значение: функция ничего не возвращает (тип `void`).

Функция вычисляет MD5-хеш для файла по указанному пути и записывает его в строковый буфер в шестнадцатеричном формате. Если открыть файл или инициализировать алгоритм MD5 не удается, функция логирует ошибку и завершает программу.

2.2.8 Функция scan_directory

Функция принимает параметры:

– `const char *dir_path` — путь к директории, которую нужно сканировать.

Возвращаемое значение: функция ничего не возвращает (тип `void`).

Функция рекурсивно сканирует указанную директорию, обрабатывая все файлы и поддиректории. Для каждого файла она вычисляет его MD5-хеш, проверяет наличие дубликатов, записывает информацию о новых файлах и удаляет ненужные дубликаты, создавая жесткие ссылки на основные файлы, если это необходимо.

2.2.9 Функция remove_files

Функция remove_files не принимает параметров и ничего не возвращает (тип void).

Функция удаляет файлы блокировки.

2.2.10 Функция signal_handler

Функция принимает параметры:

– `int signal` — номер полученного сигнала.

Возвращаемое значение: функция ничего не возвращает (тип `void`).

Функция обрабатывает различные сигналы, записывая соответствующее сообщение в лог и выполняя действия по завершению работы программы. В зависимости от сигнала (SIGINT, SIGTERM, SIGSEGV, SIGABRT), она закрывает файлы и завершает программу с кодом успеха или ошибки.

2.2.11 Функция daemonize

Функция daemonize не принимает параметров и ничего не возвращает (тип `void`).

Функция переводит процесс в фоновый режим, создавая новый сеанс и разрывая связь с управляющим терминалом, после чего перенаправляет стандартные файловые дескрипторы на /dev/null.

2.2.12 Функция can_create_lock_file

Функция не принимает никаких параметров.

Возвращаемое значение: функция возвращает CREATE_ERROR при ошибке, либо 0 при успешной проверке.

Функция проверяет возможность создания файла блокировки, пытаясь создать его с уникальным именем. Если файл не удается создать, функция выводит соответствующее сообщение об ошибке и возвращает код ошибки; если создание успешно, функция закрывает файл и возвращает 0.

2.2.13 Функция create_lock_file

Функция create_lock_file не принимает никаких параметров.

Возвращаемое значение: функция возвращает дескриптор файла блокировки при успешном создании, либо CREATE_ERROR в случае ошибки.

Функция create_lock_file создает или открывает файл блокировки, записывает в него PID текущего процесса и возвращает дескриптор этого

файла. Если не удается создать или записать файл, функция выводит ошибку и возвращает код ошибки.

2.2.14 Функция handle_kill

Функция не принимает параметров и ничего не возвращает (тип `void`).

Функция handle_kill читает PID из файла блокировки и отправляет сигнал SIGTERM процессу с этим PID. Если файл блокировки не найден, или при чтении PID или отправке сигнала возникают ошибки, функция выводит сообщение об ошибке и завершает работу.

2.2.15 Функция clean_bin_file

Функция не принимает параметров и ничего не возвращает (тип `void`).

Функция clean_bin_file открывает бинарный файл для записи в режиме "только для записи" (создавая его или обрезая существующий) и затем немедленно закрывает его, тем самым очищая его содержимое. Если файл не удается открыть, функция записывает сообщение об ошибке в лог и завершает программу.

3 РАЗРАБОТКА ПРОГРАММЫ

В данном разделе рассмотрены описания алгоритмов функций, используемых в программе.

3.1 Алгоритм функции main

Шаг 1. Начало.

Шаг 2. Получение аргументов командной строки argc и argv.

Шаг 3. Условный оператор: Первый элемент массива строк не равен «-kill». Если да, то переход к шагу 5, иначе к шагу 4.

Шаг 4. Попытка получения pid и завершения фонового процесса. Переход к шагу 14.

Шаг 5. Тестовое создание .lock файла для проверки прав.

Шаг 6. Условный оператор: .lock создан. Если да, то переход к шагу 7, иначе к шагу 14.

Шаг 7. Установка директории для сканирования.

Шаг 8. Установка обработчиков сигналов и директории для сканирования.

Шаг 9. Перевод программы в фоновый режим и запись pid в .lock.

Шаг 10. Логирование о начале сканирования и создание бинарного файла для данных о файлах. Переход к шагу 13.

Шаг 11. Сканирование директории и замена одинаковых файлов жесткими ссылками.

Шаг 12. Логирование о завершении сканирования.

Шаг 13. Остановка потока перед следующим сканированием.

Шаг 14. Конец.

3.2 Алгоритм функции find_duplicate

Шаг 1. Начало.

Шаг 2. Открытие бинарного файла.

Шаг 3. Чтение данных из файла в память.

Шаг 4. Условный оператор: Данные совпадают с переданными. Если да, то переход к шагу 6, иначе к шагу 5.

Шаг 5. Освобождение памяти и сдвиг в файле.

Шаг 6. Получение из памяти пути к файлу и возвращение его в качестве результата. Переход к шагу 8.

Шаг 7. Условный оператор: Указатель на конце файла. Если да, то переход к шагу 8, иначе к шагу 2.

Шаг 8. Конец.

3.3 Алгоритм функции compute_md5

Шаг 1. Начало.
Шаг 2. Инициализация буфера для хранения хеша и его длины.
Шаг 3. Открытие необходимого файла в бинарном режиме.
Шаг 4. Чтение данных из файла в память.
Шаг 5. Передача данных в хеш-функцию.
Шаг 6. Освобождение данных из памяти и сдвиг в файле.
Шаг 7. Условный оператор: Указатель на конце файла. Если да, то переход к шагу 8, иначе к шагу 4.
Шаг 8. Запрос на получение результата хеш функций и запись в буфер.
Шаг 9. Преобразование данных из буфера в строку в виде шестнадцатеричных символов.
Шаг 10. Возврат строки в качестве результата.
Шаг 11. Конец.

4 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

4.1 Системные требования

Для запуска данной программы необходим персональный компьютер с установленной любой UNIX-системой подобной Linux, в частности практически любым дистрибутивом Linux. На нем так же должен быть установлен пакет с библиотекой OpenSSL.

4.2 Использование приложения

Для запуска процесса необходимо предварительно его скомпилировать в консоли с помощью файла make. Для файла make установлены следующие рецепты:

– make — компилирование исходных фалов и создание необходимых директорий;

– make install — копирует файл в директорию для поиска исполняемых файлов;

– make uninstall — удаляет файл из директории поиска исполняемых файлов;

– make clean — очищает build и другие связанные с приложением директории;

Пример компиляции приложения:

```
artem@fedora:~/course_work$ make
gcc -o build/fdclean src/main.c-lssl -lcrypto \
    -DLOG_FILE=\"/home/artem/.cache/fdclean/fdclean.log\"  \
    -DLOCK_FILE=\"/home/artem/.cache/fdclean/fdclean.lock\"  \
    -DBIN_FILE=\"/home/artem/.cache/fdclean/fdclean.bin\"  \
    -DSCAN_DELAY_SEC=20
artem@fedora:~/course_work$ make install
sudo cp build/fdclean /usr/local/bin/fdclean
артем@fedora:~/course_work$
```

После компиляции необходимо выполнить следующую команду:

```
$ fdclean [директория]
```

После выполнения данной команды, программа создаст фоновый процесс, который будет проверять файлы и заменять дубликаты на жёсткие ссылки.

Пример выполнения данной команды:

```
артем@fedora:~/course_work$ fdclean test/
артем@fedora:~/course_work$
```

Для остановки фонового процесса необходимо прописать с командной строке следующую команду:

```
$ fdclean -kill
```

После этой команды, если фоновый процесс существовал, он будет отключен.

```
артем@fedora:~/course_work$ fdclean -kill
артем@fedora:~/course_work$
```

В случае, если фоновый процесс уже запущен, то при попытке создания нового процесса в консоль будет выведена ошибка. В то же время, если процесс отсутствует, то при попытке его удаления так же будет выведена ошибка.

```
артем@fedora:~/course_work$ fdclean -kill
fdclean: Процесс не запущен.
артем@fedora:~/course_work$ fdclean test/
артем@fedora:~/course_work$ fdclean ../
fdclean: Процесс уже запущен.
артем@fedora:~/course_work$
```

Для получения результатов выполнения программы необходимо зайти в директорию `.cache/fdclean/` и найти там файл `fdclean.log`. В него записываются все основные события приложения такие как начало и конец обработки директории, удаление и замена файлов на ссылки, а так же все возможные варианты ошибок и остановок процесса.

Пример log-файла:

```
[Tue Sep 17 16:23:57 2024] [INFO] Начало сканирования директории test/
[Tue Sep 17 16:23:57 2024] [INFO] Замена дубликата: удален test//sub_test/sub_sub_dir/copy1 и создана жесткая ссылка на test//copyorig
[Tue Sep 17 16:23:57 2024] [INFO] Замена дубликата: удален test//sub_test/copy2 и создана жесткая ссылка на test//copyorig
[Tue Sep 17 16:23:57 2024] [INFO] Замена дубликата: удален test//sub_test/copy3 и создана жесткая ссылка на test//copyorig
[Tue Sep 17 16:23:57 2024] [INFO] Директория test/ была успешно отсканирована, следующее сканирование через 20 сек
[Tue Sep 17 16:24:07 2024] [INFO] Получен сигнал SIGTERM, завершение работы
```

ЗАКЛЮЧЕНИЕ

В результате работы над данным курсовым проектом было разработано работоспособное приложение со своим набором функций и интерфейсом.

Работа была разделена на такие этапы, как анализ существующих аналогов, литературных источников, постановка требований к проектируемому программному продукту, системное и функциональное проектирование, разработка программных модулей и тестирование проекта. После последовательного выполнения вышеперечисленных этапов разработки было получено исправно работающее приложение.

Созданная утилита выполняет следующие функции:

– ввод директории пользователем;
– создание фонового процесса для проверки файлов;
– удаление фонового процесса;
– замена дублирующегося файла жёсткой ссылкой;
– логирование результатов фонового процесса;
– контроль за появлением дубликатов в файловой системе.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1]. Документация по Linux [Электронный ресурс]. – Режим доступа: https://linux.die.net/.

[2]. Opennet – форума с большим количеством информации о Linux [Электронный ресурс]. – Режим доступа: https://www.opennet.ru.

[3]. Хабр – информационный портал для разработчиков [Электронный ресурс]. – Режим доступа: https://habr.com/ru.

ПРИЛОЖЕНИЕ А
(Обязательное)

Блок-схема алгоритма функции main

ПРИЛОЖЕНИЕ Б
(Обязательное)

Блок-схема алгоритма функции find_duplicate

ПРИЛОЖЕНИЕ В
(Обязательное)

Блок-схема алгоритма функции compute_md5

ПРИЛОЖЕНИЕ Г
(Обязательное)

Код программы

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <openssl/evp.h>
#include <unistd.h>
#include <time.h>
#include <fcntl.h>
#include <syslog.h>
#include <signal.h>
#include <errno.h>

#define CREATE_ERROR -3      // Ошибка создания lock файла
#define FILE_IS_CLOSED -1    // Флаг закрытия файла
#define HASH_SIZE 33         // Длина строки для MD5 хеша
#define MESSAGE_BUF_SIZE 700 // Размер буфера для сообщений

#ifndef LOG_FILE
#define LOG_FILE "/home/artem/.cache/fdclean/fdclean.log" // Путь к лог-файлу
#endif

#ifndef LOCK_FILE
#define LOCK_FILE "/home/artem/.cache/fdclean/fdclean.lock" // Путь к файлу
блокировки
#endif

#ifndef BIN_FILE
#define BIN_FILE "/home/artem/.cache/fdclean/fdclear.bin" // Путь к бинарному
файлу
#endif

#ifndef DEFAULT_DIR
#define DEFAULT_DIR "/home/artem/" // Директория по умолчанию
#endif

#ifndef SCAN_DELAY_SEC
#define SCAN_DELAY_SEC 30 // Задержка до следующего сканирования
#endif

int lock_fd = FILE_IS_CLOSED; // Дескриптор файла блокировки

typedef struct FileInfo // Структура для хранения информации о файле
{
    off_t pathLength;     // Длина пути к файлу
    char *path;           // Путь к файлу
    off_t size;           // Размер файла
    char hash[HASH_SIZE]; // MD5 хеш
} FileInfo;

void log_message(const char *, const char *);
int check_same_inode(const char *path1, const char *path2);
void write_file_info(const FileInfo *);
char *find_duplicate(const FileInfo *);
void compute_md5(const char *, char *);
void scan_directory(const char *);
void remove_files();
void signal_handler(int);
void daemonize();
int can_create_lock_file();
int create_lock_file();
```

```
void handle_kill();
void clean_bin_file();

// Основная функция
int main(int argc, char *argv[])
{
    if (argc > 1)
    {
        if (strcmp(argv[1], "-kill") == 0)
        {
            handle_kill();
        }
    }

    if (can_create_lock_file() == CREATE_ERROR)
    {
        return EXIT_FAILURE;
    }

    const char *directory = (argc > 1) ? argv[1] : DEFAULT_DIR;

    // Устанавливаем обработчик сигнала
    signal(SIGTERM, signal_handler);
    signal(SIGINT, signal_handler);
    signal(SIGSEGV, signal_handler);
    signal(SIGABRT, signal_handler);

    daemonize(); // Переводим программу в фоновый режим

    lock_fd = create_lock_file();
    if (lock_fd < 0)
    {
        remove_files();
        return EXIT_FAILURE;
    }

    char message_start[MESSAGE_BUF_SIZE];
    char message_end[MESSAGE_BUF_SIZE];
    snprintf(message_start,     MESSAGE_BUF_SIZE,     "Начало     сканирования
директории %s", directory);
    snprintf(message_end,  MESSAGE_BUF_SIZE,  "Директория  %s  была  успешно
отсканирована,   следующее   сканинование   через   %d   сек",   directory,
SCAN_DELAY_SEC);
    while (1)
    {

        log_message("INFO", message_start);
        clean_bin_file();
        scan_directory(directory);
        log_message("INFO", message_end);
        sleep(SCAN_DELAY_SEC);
    }

    remove_files();
    return 0;
}

// Функция для логирования
void log_message(const char *type, const char *message)
{
    FILE *log_file = fopen(LOG_FILE, "a");
    if (!log_file)
```

```
    {
        syslog(LOG_ERR, "Ошибка открытия лог-файла");
        remove_files();
        exit(EXIT_FAILURE);
    }

    // Получаем текущее время для записи в лог
    time_t now = time(NULL);
    char *time_str = ctime(&now);
    time_str[strlen(time_str) - 1] = '\0'; // Удаляем символ новой строки

    // Записываем сообщение в лог с указанием типа (INFO или ERROR)
    fprintf(log_file, "[%s] [%s] %s\n", time_str, type, message);
    fclose(log_file);
}

int check_same_inode(const char *path1, const char *path2)
{
    struct stat stat1, stat2;

    // Получаем информацию о первом файле
    if (stat(path1, &stat1) == -1)
    {
        char error_message[MESSAGE_BUF_SIZE];
        snprintf(error_message,     MESSAGE_BUF_SIZE,     "Ошибка     получения
информации о файле %s: %s", path1, strerror(errno));
        log_message("ERROR", error_message);
        remove_files();
        exit(EXIT_FAILURE);
    }

    // Получаем информацию о втором файле
    if (stat(path2, &stat2) == -1)
    {
        char error_message[MESSAGE_BUF_SIZE];
        snprintf(error_message,     MESSAGE_BUF_SIZE,     "Ошибка     получения
информации о файле %s: %s", path2, strerror(errno));
        log_message("ERROR", error_message);
        remove_files();
        exit(EXIT_FAILURE);
    }
    // Сравниваем номера inode
    return (stat1.st_ino == stat2.st_ino && stat1.st_dev == stat2.st_dev);
}

// Функция для записи информации о файле в бинарный файл
void write_file_info(const FileInfo *file_info)
{
    FILE *file = fopen(BIN_FILE, "ab"); // Открываем файл в режиме добавления
(бинарный)
    if (!file)
    {
        log_message("ERROR", "Ошибка чтения бинарного файла");
        remove_files();
        exit(EXIT_FAILURE);
    }

    // Записываем данные
    fwrite(&file_info->pathLength, sizeof(file_info->pathLength), 1, file);
    fwrite(file_info->path, 1, file_info->pathLength, file);
    fwrite(&file_info->size, sizeof(file_info->size), 1, file);
    fwrite(file_info->hash, 1, HASH_SIZE, file);
```

```
    fclose(file);
}

// Функция для удаления информации о файле из бинарного файла по пути
void delete_info_by_path(const char *search_path)
{
    FILE *file = fopen(BIN_FILE, "rb");
    if (!file)
    {
        log_message("ERROR", "Ошибка открытия бинарного файла для чтения");
        remove_files();
        exit(EXIT_FAILURE);
    }

    // Создаем временный файл для записи обновленного содержимого
    FILE *temp_file = fopen(BIN_FILE ".tmp", "wb");
    if (!temp_file)
    {
        log_message("ERROR", "Ошибка открытия временного файла для записи");
        fclose(file);
        remove_files();
        exit(EXIT_FAILURE);
    }

    size_t path_length;
    char *path;
    int found = 0;

    while (fread(&path_length, sizeof(path_length), 1, file))
    {
        path = malloc(path_length);
        if (!path)
        {
            log_message("ERROR", "Ошибка выделения памяти");
            fclose(file);
            fclose(temp_file);
            remove_files();
            exit(EXIT_FAILURE);
        }

        fread(path, 1, path_length, file);
        off_t file_size;
        fread(&file_size, sizeof(file_size), 1, file);
        char file_hash[HASH_SIZE];
        fread(file_hash, 1, HASH_SIZE, file);

        // Записываем в временный файл, если путь не соответствует искомому
        if (strcmp(path, search_path) != 0)
        {
            fwrite(&path_length, sizeof(path_length), 1, temp_file);
            fwrite(path, 1, path_length, temp_file);
            fwrite(&file_size, sizeof(file_size), 1, temp_file);
            fwrite(file_hash, 1, HASH_SIZE, temp_file);
        }
        else
        {
            found = 1; // Помечаем, что удаление произошло
        }

        free(path); // Освобождаем выделенную память для пути
    }
```

```
    fclose(file);
    fclose(temp_file);

    if (remove(BIN_FILE) != 0)
    {
        log_message("ERROR", "Ошибка удаления оригинального бинарного файла");
        return;
    }

    if (rename(BIN_FILE ".tmp", BIN_FILE) != 0)
    {
        log_message("ERROR", "Ошибка переименования временного файла");
        return;
    }

    if (found)
    {
        // Логируем успешное удаление данных о файле
        char message[MESSAGE_BUF_SIZE];
        snprintf(message, MESSAGE_BUF_SIZE, "Успешное удаление данных о файле
с путем %s", search_path);
        log_message("INFO", message);
    }
}

// Функция для проверки на дубликаты перед записью
char *find_duplicate(const FileInfo *new_file_info)
{
    FILE *file = fopen(BIN_FILE, "rb");
    if (!file)
    {
        log_message("ERROR", "Ошибка открытия файла для чтения");
        remove_files();
        exit(EXIT_FAILURE);
    }

    FileInfo file_info;
    while   (fread(&file_info.pathLength,   sizeof(file_info.pathLength),   1,
file))
    {
        file_info.path = malloc(file_info.pathLength);
        if (!file_info.path)
        {
            fclose(file);
            log_message("ERROR", "Ошибка выделения памяти");
            remove_files();
            exit(EXIT_FAILURE);
        }

        fread(file_info.path, 1, file_info.pathLength, file);
        fread(&file_info.size, sizeof(file_info.size), 1, file);
        fread(file_info.hash, 1, HASH_SIZE, file);

        if (file_info.size == new_file_info->size &&
            memcmp(file_info.hash, new_file_info->hash, HASH_SIZE) == 0)
        {
            fclose(file);
            return file_info.path; // Возвращаем путь к дубликату
        }

        free(file_info.path);
```

```
    }
    fclose(file);
    return NULL; // Нет дубликата
}

// Функция для вычисления MD5 хеша файла
void compute_md5(const char *path, char *hash_str)
{
    unsigned char hash[EVP_MAX_MD_SIZE]; // Буфер для хеша
    unsigned int hash_len;
    EVP_MD_CTX *mdctx = EVP_MD_CTX_new();

    FILE *file = fopen(path, "rb");
    if (!file)
    {
        char message[MESSAGE_BUF_SIZE];
        snprintf(message, MESSAGE_BUF_SIZE, "Ошибка открытия файла для
хеширования: %s", path);
        log_message("ERROR", message);
        EVP_MD_CTX_free(mdctx);
        fclose(file);
        remove_files();
        exit(EXIT_FAILURE);
    }

    if (EVP_DigestInit_ex(mdctx, EVP_md5(), NULL) != 1)
    {
        log_message("ERROR", "Ошибка инициализации MD5");
        EVP_MD_CTX_free(mdctx);
        fclose(file);
        remove_files();
        exit(EXIT_FAILURE);
        return;
    }

    char buf[MESSAGE_BUF_SIZE];
    int bytes;
    while ((bytes = fread(buf, 1, MESSAGE_BUF_SIZE, file)) > 0)
    {
        EVP_DigestUpdate(mdctx, buf, bytes);
    }
    fclose(file);

    EVP_DigestFinal_ex(mdctx, hash, &hash_len);
    EVP_MD_CTX_free(mdctx);

    for (unsigned int i = 0; i < hash_len; i++)
    {
        sprintf(&hash_str[i * 2], "%02x", hash[i]);
    }
    hash_str[HASH_SIZE - 1] = '\0'; // Завершающий ноль
}

// Функция для рекурсивного обхода директории и записи уникальных файлов
void scan_directory(const char *dir_path)
{
    DIR *dir = opendir(dir_path);
    if (!dir)
    {
        fprintf(stderr, "fdclean: Ошибка открытия директории: %s", dir_path);
        remove_files();
        exit(EXIT_FAILURE);
```

```
    }

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL)
    {
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") ==
0)
        {
            continue;
        }

        // Динамическое выделение памяти для пути
        size_t path_length = strlen(dir_path) + strlen(entry->d_name) + 2; //
+2 для '/' и '\0'
        char *path = malloc(path_length);
        if (!path)
        {
            fprintf(stderr, "fdclean: Ошибка выделения памяти");
            closedir(dir);
            remove_files();
            exit(EXIT_FAILURE);
        }

        snprintf(path, path_length, "%s/%s", dir_path, entry->d_name);

        struct stat statbuf;
        if (stat(path, &statbuf) == 0)
        {
            if (S_ISDIR(statbuf.st_mode))
            {
                scan_directory(path); // Рекурсивно сканируем директорию
            }
            else if (S_ISREG(statbuf.st_mode))
            {
                char hash[HASH_SIZE];
                compute_md5(path, hash);

                FileInfo file_info;
                file_info.pathLength = strlen(path) + 1;
                file_info.path = malloc(file_info.pathLength);
                if (!file_info.path)
                {
                    fprintf(stderr, "fdclean: Ошибка выделения памяти");
                    free(path);
                    remove_files();
                    exit(EXIT_FAILURE);
                }
                strcpy(file_info.path, path);
                file_info.size = statbuf.st_size;
                memcpy(file_info.hash, hash, HASH_SIZE);
                char *file_duplicate_path = find_duplicate(&file_info);
                // Проверка на дубликаты
                if (file_duplicate_path == NULL)
                {
                    write_file_info(&file_info);  //  Записываем информацию  о
файле
                }
                else
                {
                    if                 (!check_same_inode(file_duplicate_path,
file_info.path))
                    {
```

```
                        // Удаляем дубликат
                        if (unlink(file_info.path) == -1)
                        {
                            fprintf(stderr,    "fdclean:    Ошибка    удаления
дубликата %s: %s", file_info.path, strerror(errno));
                            remove_files();
                            exit(EXIT_FAILURE);
                        }
                        else
                        {
                            // Создаем жесткую ссылку на основной файл
                            if  (link(file_duplicate_path,  file_info.path)  ==
-1)
                            {
                                fprintf(stderr,   "fdclean:   Ошибка   создания
жесткой ссылки на %s: %s", file_info.path, strerror(errno));
                                remove_files();
                                exit(EXIT_FAILURE);
                            }
                            else
                            {
                                char message[MESSAGE_BUF_SIZE * 2];
                                snprintf(message,   MESSAGE_BUF_SIZE   *   2,
"Замена дубликата: удален %s и создана жесткая ссылка на %s", file_info.path,
file_duplicate_path);
                                log_message("INFO", message);
                            }
                        }
                    }

                    free(file_duplicate_path);   //   Освобождаем   выделенную
память
                }

                free(file_info.path); // Освобождаем выделенную память
            }
        }

        free(path); // Освобождаем выделенную память
    }
    closedir(dir);
}

// Функция для удаления файла блокировки
void remove_files()
{
    if (lock_fd != FILE_IS_CLOSED)
    {
        close(lock_fd);
    }
    unlink(LOCK_FILE);
    unlink(BIN_FILE);
}

// Обработчик сигнала завершения демона
void signal_handler(int signal)
{
    switch (signal)
    {
    case SIGINT:
        log_message("INFO",  "Получен  сигнал  SIGINT  (Ctrl+C),  завершение
работы");
```

```
            remove_files();
            exit(EXIT_SUCCESS);
            break;
        case SIGTERM:
            log_message("INFO", "Получен сигнал SIGTERM, завершение работы");
            remove_files();
            exit(EXIT_SUCCESS);
            break;
        case SIGSEGV:
            log_message("ERROR", "Ошибка сегментации (SIGSEGV). Завершение работы
программы.");
            remove_files();
            exit(EXIT_FAILURE);
            break;
        case SIGABRT:
            log_message("ERROR",  "Принудительный  сброс  (SIGABRT).  Завершение
работы программы.");
            remove_files();
            exit(EXIT_FAILURE);
            break;
        default:
            break;
        }
}

// Функция для создания демона
void daemonize()
{

    pid_t pid = fork();
    if (pid < 0)
    {
        exit(EXIT_FAILURE);
    }

    if (pid > 0)
    {
        exit(EXIT_SUCCESS); // Родительский процесс завершает выполнение
    }

    if (setsid() < 0)
    {
        exit(EXIT_FAILURE);
    }

    // Перекрываем сигналы SIGHUP и SIGCHLD
    signal(SIGHUP, SIG_IGN);
    signal(SIGCHLD, SIG_IGN);

    // fork для защиты от повторной инициализации терминала
    pid = fork();

    if (pid < 0)
    {
        exit(EXIT_FAILURE);
    }

    if (pid > 0)
    {
        exit(EXIT_SUCCESS); // Родительский процесс завершает выполнение
    }
```

```
    // Закрываем все файловые дескрипторы
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);

    // Открываем стандартные файловые дескрипторы на /dev/null
    open("/dev/null", O_RDONLY); // stdin
    open("/dev/null", O_RDWR);   // stdout
    open("/dev/null", O_RDWR);   // stderr
}

// Функция для проверки, можно ли создать файл блокировки
int can_create_lock_file()
{
    int fd = open(LOCK_FILE, O_CREAT | O_EXCL | O_WRONLY, 0600);
    if (fd < 0)
    {
        if (errno == EEXIST)
        {
            fprintf(stderr, "fdclean: Процесс уже запущен.\n");
        }
        else if (errno == EACCES)
        {
            fprintf(stderr, "fdclean: Недостаточно прав для создания .lock
файла.\n");
        }
        else if (errno == EROFS)
        {
            fprintf(stderr, "fdclean: Файловая система доступна только для
чтения.\n");
        }
        else
        {
            fprintf(stderr, "fdclean: Ошибка создания .lock файла: ");
            perror(NULL); // Выводит описание ошибки, основанное на значении
errno
        }
        return CREATE_ERROR;
    }
    close(fd); // Закрываем файл, т.к. проверка пройдена
    return 0;  // Успешная проверка
}

// Функция для создания файла блокировки и записи PID
int create_lock_file()
{
    int fd = open(LOCK_FILE, O_CREAT | O_WRONLY | O_TRUNC, 0600);
    if (fd < 0)
    {
        perror("fdclean: Ошибка открытия .lock файла для записи.");
        return CREATE_ERROR;
    }

    // Записываем PID в файл блокировки
    char pid_str[20];

    snprintf(pid_str, sizeof(pid_str), "%d\n", getpid());
    if (write(fd, pid_str, strlen(pid_str)) != strlen(pid_str))
    {
        perror("fdclean: Ошибка записи PID в .lock файл.");
        close(fd);
        return CREATE_ERROR;
```

```
    }

    return fd; // Возвращаем дескриптор файла блокировки
}

void handle_kill()
{
    FILE *lock_file = fopen(LOCK_FILE, "r");
    if (lock_file == NULL)
    {
        fprintf(stderr, "fdclean: Процесс не запущен.\n");
        exit(EXIT_FAILURE);
    }

    int pid;
    if (fscanf(lock_file, "%d", &pid) != 1)
    {
        fprintf(stderr, "fdclean: Ошибка чтения PID из файла блокировки.\n");
        fclose(lock_file);
        exit(EXIT_FAILURE);
    }
    fclose(lock_file);

    // Отправляем сигнал SIGTERM процессу
    if (kill(pid, SIGTERM) != 0)
    {
        if (errno == ESRCH)
        {
            fprintf(stderr, "fdclean: Процесс не найден.\n", pid);
        }
        else
        {
            perror("fdclean: Ошибка при попытке остановить процесс");
        }
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}

void clean_bin_file()
{
    FILE *file = fopen(BIN_FILE, "wb");
    if (!file)
    {
        log_message("ERROR", "fdclean: Ошибка открытия бинарного файла");
        remove_files();
        exit(EXIT_FAILURE);
    }
    fclose(file);
}
```