

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 01 002 ПЗ

Студент:

Власов Р. Е

Руководитель:

старший преподаватель
кафедры ЭВМ
Л.П. Поденок

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1 ОБЗОР ЛИТЕРАТУРЫ.....	4
2 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ	8
3 РАЗРАБОТКА ПРОГРАММЫ	12
4 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ.....	14
ЗАКЛЮЧЕНИЕ	17
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ.....	18
ПРИЛОЖЕНИЕ А	19
ПРИЛОЖЕНИЕ Б	20
ПРИЛОЖЕНИЕ В.....	21
ПРИЛОЖЕНИЕ Г	22

ВВЕДЕНИЕ

Современные файловые системы часто сталкиваются с проблемой наличия идентичных файлов, что приводит к нерациональному расходованию дискового пространства и снижению общей производительности системы. Инструменты для обнаружения и устранения таких дублированных файлов являются ключевыми для эффективной организации файловых хранилищ.

Одним из эффективных способов решения данной задачи является использование жёстких ссылок — особого вида файловых ссылок, благодаря которым несколько записей могут ссылаться на один и тот же физический файл на диске, исключая тем самым необходимость хранения нескольких копий данных. Это позволяет существенно экономить дисковое пространство и одновременно повышать производительность за счёт уменьшения нагрузки на файловую систему при обработке повторяющихся данных. Кроме того, такой подход минимизирует вероятность ошибок, возникающих при ручном удалении или перемещении файлов, и обеспечивает поддержание целостности и согласованности данных даже при интенсивных изменениях в файловой системе.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Обзор предметной области

Проблема избыточного копирования файлов знакома как частным пользователям, так и крупным организациям. По мере роста объёмов данных на серверах и персональных устройствах одни и те же файлы нередко оказываются сохранёнными в разных каталогах, что приводит к неоправданному занятию дискового пространства и замедлению работы системы. В условиях ограниченных ресурсов — будь то ёмкость хранилища или вычислительная мощность — грамотная организация хранения данных становится приоритетом.

Одним из наиболее надёжных приёмов оптимизации хранилищ является применение жёстких ссылок. Жёсткая ссылка позволяет «привязать» несколько записей к одному и тому же блоку на диске, избавляясь от необходимости хранения повторяющихся копий, при этом сохраняя прямой доступ к содержимому.

В отличие от символических ссылок, жёсткие ссылки не теряют работоспособности при перемещении или удалении исходного файла — до тех пор, пока существует хотя бы одна ссылка на данные, они остаются доступными. Данная технология поддерживается многими файловыми системами: HFS+ и APFS на macOS, ext4 в Linux, NTFS в Windows и другими, что делает её универсальным инструментом для повышения эффективности использования дискового пространства.

Не менее важным элементом подобных решений является ведение журнала операций по замене дубликатов жёсткими ссылками. Протоколирование обеспечивает:

- фиксацию сведений о найденных дубликатах и выполненных преобразованиях;
- регистрацию ошибок и исключительных ситуаций
- возможность восстановления исходного состояния при необходимости;
- гибкую настройку уровня детализации логов

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 Duplicate File Finder

Duplicate File Finder– утилита, ориентированная на поиск и управление дубликатами файлов с продвинутыми возможностями и графическим интерфейсом. Интерфейс и частичный функционал изображены на рисунке 1.2.2.

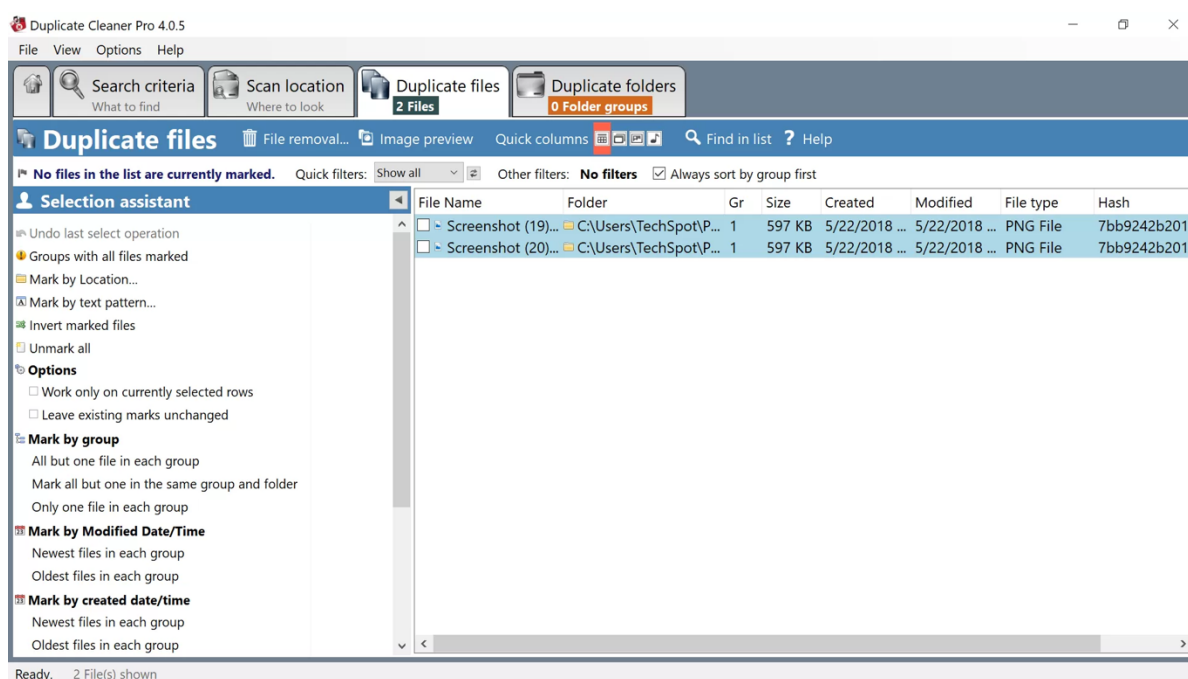


Рисунок 1.2.2 – Скриншот программы Duplicate Cleaner

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.

1.2.3 Duff

Duff — 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 1.2.3.

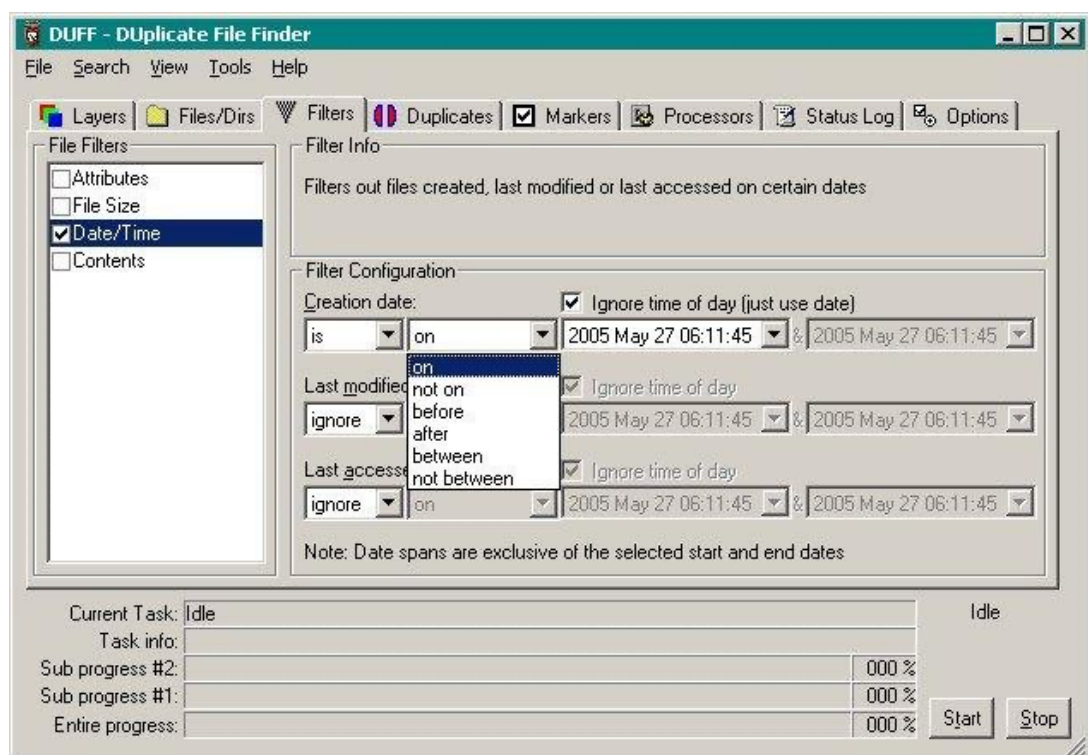


Рисунок 1.2.3 – Скриншот программы Duff

Основные функции Duff:

- 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.

1.2.4 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.

1.3 Постановка задачи

Анализ существующих решений показал, что они включают множество функций, реализация которых выходит за рамки курсового проекта. В связи с этим для выполнения в течение семестра были отобраны ключевые возможности:

- ввод директории пользователем;
- организация фонового процесса для проверки файлов на дубликаты;
- возможность корректного завершения фонового процесса;
- замена найденных дубликатов жёсткими ссылками;
- логирование результатов фонового процесса;
- постоянный мониторинг файловой системы на предмет появления новых дубликатов.

2 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описывается функционирование и структура разрабатываемого приложения.

2.1 Структура входных и выходных данных

Входные данные передаются через аргументы командной строки:

- `kill`. При наличии этого аргумента программа читает PID из файла блокировки (`LOCK_FILE`) и посылает соответствующему процессу сигнал `SIGTERM` для корректного завершения процесса.
- путь к каталогу для сканирования. Если указан один аргумент без ключа `-kill`, он рассматривается как абсолютный или относительный путь к директории, которую следует обходить в поисках дубликатов. (`DEFAULT_DIR`).

Если аргументы не переданы, утилита по умолчанию сканирует каталог пользователя (`DEFAULT_DIR`), заданный через макрос (обычно `$HOME`).

Выходные данные — это файл журнала (`LOG_FILE`), в который приложение записывает информацию о своей работе:

- метки времени и уровень сообщений (`INFO / ERROR`);
- операции запуска и завершения каждого прохода сканирования;
- информация о найденных и заменённых дубликатах (какой файл был заменён на жёсткую ссылку);
- сообщения об ошибках при открытии каталогов, файлов или при создании ссылок.

2.2 Описание работы функций

2.2.1 Функция `main`

Функция принимает параметры:

- `int argc` — число аргументов командной строки (включая имя программы);
- `char *argv[]` — массив строк, где `argv[0]` — путь к исполняемому файлу, а `argv[1...argc-1]` — переданные пользователем параметры.

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

В начале выполнения `main` считывает из окружения переменную `HOME` и формирует на её основе пути к каталогу кеша, файлам логирования, блокировки и двоичному журналу. Если обнаружен аргумент `-kill`, вызывается функция остановки активного процесса и программа завершается.

В противном случае создаётся нужный каталог кеша, проверяется отсутствие уже запущенного экземпляра через файл блокировки, настраиваются обработчики сигналов. Затем создаётся lock-файл с текущим PID, после чего в бесконечном цикле с паузой последовательно выполняются: запись в лог сообщения о старте обхода, очистка двоичного журнала, рекурсивный просчёт MD5 и замена дубликатов жёсткими ссылками, и запись в лог информации об окончании прохода.

2.2.2 Функция `log_message`

Функция принимает два параметра: строку уровня сообщения ("INFO" или "ERROR") и собственно текст сообщения. Она открывает лог-файл в режиме добавления, формирует метку времени в заданном формате даты, затем записывает в файл строку вида [timestamp] [level] message. При невозможности открыть файл для записи производится запись об ошибке через syslog, выполняется очистка временных файлов и программа завершается, что гарантирует консистентность lock-файла и бинарного журнала.

2.2.3 Функция `make_path`

Функция `make_path` служит для безопасного формирования строковых путей: она принимает указатель на буфер `dst` и его размер `dtsz`, формат строки `fmt` и строку `home`, содержащую путь к домашнему каталогу. Внутри один вызов `snprintf(dst, dtsz, fmt, home)` заполняет буфер корректной NUL-терминированной строкой. Возвращаемое значение отсутствует.

2.2.4 Функция `ensure_singleton`

Получает путь к файлу блокировки и пытается создать его с флагами `O_CREAT`, `O_EXCL`, `O_WRONLY`. Если файл уже существует (`errno == EEXIST`), она печатает в `stderr` сообщение: уже запущен и возвращает `-1`; при других ошибках использует `perror`, также возвращая `-1`. В случае успешного создания сразу закрывает дескриптор и удаляет этот пробный lock-файл (чтобы не мешал настоящему), возвращая `0`.

2.2.5 Функция `create_lock_file`

При старте демона эта функция заново создаёт lock-файл (флаги `O_WRONLY`, `O_CREAT`, `O_TRUNC`), записывает в него PID процесса через `snprintf` и `write` и возвращает открытый файловый дескриптор. В случае ошибки открытия возвращает `-1` (после вызова `perror("lock create")`).

2.2.6 Функция `kill_running`

Для аргумента `-kill` используется `kill_running`, которая открывает файл блокировки в режиме чтения, извлекает из него целочисленный PID через `fscanf` и отправляет этому процессу сигнал `SIGTERM`. В случае успеха печатает `deduplar: отправлен SIGTERM "pid"`, иначе вызывает `perror`. Если открыть файл не удалось, выводит `deduplar: процесс не найден`.

2.2.7 Функция `reset_bin`

Чтобы каждый цикл начинался очищенным, `reset_bin` открывает двоичный журнал `g_bin_path` в режиме `wb` (что обнуляет его содержимое) и тут же закрывает файл. Если открытие не удалось, функция `silently` пропускает ошибку.

2.2.8 Функция `same_inode`

Для определения того, принадлежат ли два файла одной физической сущности, функция `same_inode` принимает пути `a` и `b`, выполняет `stat` для каждого и сравнивает поля `st_dev` и `st_ino`. Если оба вызова `stat` успешны и поля совпадают, возвращается ненулевое значение (истина), иначе — 0.

2.2.9 Функция `write_bin`

Каждый обнаруженный уникальный файл записывается в бинарный журнал функцией `write_bin`, которой передаётся структура `file_info_t`, содержащая длину пути, сам путь, размер файла и MD5-хеш. Открытие `g_bin_path` в режиме `ab` при сбое вызывает `log_msg("ERROR", ...)`, `cleanup()` и завершение процесса. При успехе поля структуры последовательно записываются вызовами `fwrite`, после чего файл закрывается.

2.2.10 Функция `lookup_duplicate`

Поиск ранее встреченных файлов выполняет `lookup_duplicate`, открывающая журнал `g_bin_path` в режиме `rb`. В цикле считываются сначала длина пути, затем сама строка пути (с выделением памяти), размер файла и MD5-хеш. Если размер и хеш совпадают с тем, что ищется, файл закрывается, и возвращается указатель на строку с путём дубликата (`caller` обязан освободить память).

Если записи заканчиваются, функция закрывает файл и возвращает `NULL`.

2.2.11 Функция md5_file

Вычисление контрольной суммы организовано в `md5_file`, принимающей путь `path` и буфер `out` длиной `HASH_TEXT_LEN`. Сначала создаётся контекст OpenSSL через `EVP_MD_CTX_new`, затем файл открывается в режиме `rb`. Если не удалось ни открыть файл, ни инициализировать контекст, фиксируется ошибка, контекст и/или файл закрываются, и функция возвращает управление вызывающему. При успешной инициализации выполняется `EVP_DigestInit_ex(ctx, EVP_md5(), NULL)`, затем в цикле считываются блоки по 4096 байт, передавая их в `EVP_DigestUpdate`. Финализация (`EVP_DigestFinal_ex`) заполняет буфер байт хеша и длину, контекст освобождается, файл закрывается, и каждый байт преобразуется в две шестнадцатеричные цифры через `sprintf`, с установкой завершающего нуля.

2.2.12 Функция traverse

Рекурсивный обход директорий выполняет `traverse`, открывающая каталог через `opendir(dir)`. Если открыть не удалось, вызывается `log_msg` с ошибкой и возвращается. Для каждой записи формируется путь, вызывается `stat`. Если это вложенная директория, `traverse` вызывается рекурсивно, иначе если это обычный файл, создаётся структура `file_info_t` с заполнением размера, длины пути и пути (через `strdup`), затем вычисляется MD5. После этого вызывается `lookup_duplicate`; при отсутствии дубликата — `write_bin`, при обнаружении дубликата и несовпадающем `inode` сначала удаляется текущий файл (`unlink`), затем создаётся жёсткая ссылка `link(dup, fi.path)`, а результат замены логируется через `log_msg`; при ошибке создания ссылки пишется `log_msg` с ошибкой. Память под пути освобождается, и, если был дубликат, освобождается указатель `dup`. По завершении обхода каталог закрывается через `closedir`.

2.2.13 Функция cleanup

Удаление временных артефактов и завершение работы процесса выполняет `cleanup`, закрывающая дескриптор `g_lock_fd`, если он ≥ 0 , и удаляющая файлы `g_lock_path` и `g_bin_path` через `unlink`.

2.2.14 Функция on_signal

Обработчик сигналов `on_signal` принимает номер сигнала `sig`, записывает в лог через `log_msg` соответствующее сообщение: для `SIGINT` и `SIGTERM` уровень "INFO", для `SIGSEGV` и `SIGABRT` — "ERROR". После записи выполняет `cleanup()` и завершает процесс кодом `EXIT_SUCCESS` для нормальных сигналов и `EXIT_FAILURE` для аварийных.

3 РАЗРАБОТКА ПРОГРАММЫ

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

3.1 Алгоритм функции `main`

Шаг 1. Начало.

Шаг 2. Условный оператор: если переменная среды задана, то шаг 3, иначе переход к шагу 15.

Шаг 3. Сформировать рабочие пути: каталог кэша, лог-файл, lock-файл, бинарный журнал.

Шаг 4. Условный оператор: если передан ключ «-kill», то переход к шагу 5, иначе к шагу 6.

Шаг 5. Прочитать PID из lock-файла и попытаться завершить связанный процесс; затем перейти к шагу 15.

Шаг 6. Создать при необходимости каталог кэша.

Шаг 7. Тест-создание временного lock-файла, чтобы убедиться, что приложение ещё не запущено и у есть права записи; если тест не прошёл → шаг 15.

Шаг 8. Определить директорию для сканирования (параметр пользователя или корневой каталог).

Шаг 9. Назначить обработчики системных сигналов.

Шаг 10. Перевести процесс в фон (двойной `fork`), создать постоянный lock-файл, записать в него PID.

Шаг 11. Зафиксировать в логе начало сканирования; очистить бинарный журнал файлов.

Шаг 12. Рекурсивно просканировать назначенную директорию, при необходимости заменяя дубликаты жёсткими ссылками.

Шаг 13. Записать в логе завершение прохода.

Шаг 14. Приостановиться на заданный интервал и вернуться к шагу 11.

Шаг 15. Конец.

3.2 Алгоритм функции `lookup_duplicate`

Шаг 1. Начало.

Шаг 2. Открыть бинарный журнал в режиме чтения; если отсутствует, то возвращается NULL и переход на шаг 11, если присутствует, то переход к шагу 4.

Шаг 3. Прочитать 32-битную длину следующего сохранённого пути.

Шаг 4. Проверить, удалось ли прочитать длину, если нет, то переход на шаг 6, если да, то переход на шаг 7.

Шаг 5. Закрыть файл и вернуть NULL (достигнут конец журнала либо ошибка).

Шаг 6. Выделить память под строку указанной длины и прочитать сам путь.

Шаг 7. Считать из журнала размер файла и его MD5-хэш.

Шаг 8. Условный оператор: сравнить полученные размеры и MD5 с искомыми значениями, если совпадают, то шаг 9, а если не совпадают, то шаг 10.

Шаг 9. Закрывать файл и вернуть указатель на найденный путь и переход к шагу 12.

Шаг 10. Освободить выделенную память и перейти к следующей записи (вернуться к Шагу 4).

Шаг 11. Конец.

3.3 Алгоритм функции `compute_md5`

Шаг 1. Начало.

Шаг 2. Инициализировать буфер для хранения MD5-хеша и переменную его длины.

Шаг 3. Создание контекст хеширования и открытие необходимого файла в бинарном режиме.

Шаг 4. Условный оператор: если ресурсы не получены, то занести ошибку в лог и завершить функцию.

Шаг 5. Запустить подсчёт хеша и считать очередной блок данных из файла в оперативную память.

Шаг 6. Передача считанного блока в MD5-контекст для обновления хеша.

Шаг 7. Условный оператор: Указатель на конце файла. Если да, то переход к шагу 8, иначе к шагу 6.

Шаг 8. Финализация вычислений и запись готового двоичного хеша в буфер.

Шаг 9. Закрытие файла, освобождение MD5-контекста.

Шаг 10. Преобразование данных из буфера в строку в виде шестнадцатеричных символов.

Шаг 11. Вернуть указатель на сформированную строку.

Шаг 12. Конец.

4 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

4.1 Системные требования

Для запуска данной программы необходим персональный компьютер с установленной любой UNIX-системой подобной Linux, MacOS. 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 команды:

- make — компилирование исходных файлов 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:

```
Roman@MacBook-Pro-Roman deduplar % make
mkdir -p build/ /Users/Roman/Library/Caches/deduplar
clang -o build/deduplar src/deduplar.c -Wall -Wextra -pedantic -
std=c11 -I/opt/homebrew/opt/openssl@3/include -
L/opt/homebrew/opt/openssl@3/lib -Wl,-
rpath,/opt/homebrew/opt/openssl@3/lib -lssl -lcrypto \
-DLOG_FILE=\"/Users/Roman/Library/Caches/deduplar/deduplar.log\" \
-DLOCK_FILE=\"/Users/Roman/Library/Caches/deduplar/deduplar.lock\" \
-DBIN_FILE=\"/Users/Roman/Library/Caches/deduplar/deduplar.bin\" \
-DSCAN_INTERVAL=20
Roman@MacBook-Pro-Roman deduplar % sudo make install
Password:
sudo install -m 755 build/deduplar /usr/local/bin/deduplar
Roman@MacBook-Pro-Roman deduplar %
```

После компиляции необходимо выполнить следующую команду:

```
deduplar [директория]
```

При отсутствии аргумента сканируется root директория.

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

Пример выполнения данной команды:

```
Roman@MacBook-Pro-Roman deduplar %  
deduplar ~/Desktop/test-deduplicator
```

Для проверки корректности программы можно просмотреть ссылки файлов всего каталога при помощи:

```
Roman@MacBook-Pro-Roman deduplar % ls -li ~/Desktop/test-  
deduplicator  
total 120  
265792100 -rw-r--r--@ 3 Roman  staff  12413 May 11 23:19 file1.c  
265792100 -rw-r--r--@ 3 Roman  staff  12413 May 11 23:19 file2.c  
265792100 -rw-r--r--@ 3 Roman  staff  12413 May 11 23:19 file3.c  
265792084 -rw-r--r--  3 Roman  staff      15 May 11 23:19  
fileA.txt  
265792084 -rw-r--r--  3 Roman  staff      15 May 11 23:19  
fileB.txt  
265792084 -rw-r--r--  3 Roman  staff      15 May 11 23:19  
fileC.txt
```

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

```
ddeduplar -kill
```

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

```
Roman@MacBook-Pro-Roman deduplar % deduplar -kill  
deduplar: Отправлен SIGTERM 63447 для завершения
```

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.

```
Roman@MacBook-Pro-Roman deduplar % deduplar ~/Desktop/test-  
deduplicator  
deduplar: уже запущен  
Roman@MacBook-Pro-Roman deduplar % deduplar -kill  
deduplar: Отправлен SIGTERM 63447 для завершения  
Roman@MacBook-Pro-Roman deduplar % deduplar -kill  
deduplar: Процесс не найден
```

Для получения результатов выполнения программы необходимо зайти в директорию `/var/root/Library/Caches/deduplar` и найти там файл `deduplar.log`. Либо ввести команду:
`tail -n +1 -f ~/Library/Caches/deduplar/deduplar.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:

```
[2025-05-11 23:19:20] [INFO] Начало обхода
/Users/Roman/Desktop/test-deduplicator
[2025-05-11 23:19:20] [INFO] Заменен /Users/Roman/Desktop/test-
deduplicator/fileC.txt с жесткой ссылкой на
/Users/Roman/Desktop/test-deduplicator/fileB.txt
[2025-05-11 23:19:20] [INFO] Заменен /Users/Roman/Desktop/test-
deduplicator/fileA.txt с жесткой ссылкой на
/Users/Roman/Desktop/test-deduplicator/fileB.txt
[2025-05-11 23:19:20] [INFO] Заменен /Users/Roman/Desktop/test-
deduplicator/file1.c с жесткой ссылкой на
/Users/Roman/Desktop/test-deduplicator/file2.c
[2025-05-11 23:19:20] [INFO] Заменен /Users/Roman/Desktop/test-
deduplicator/file3.c с жесткой ссылкой на
/Users/Roman/Desktop/test-deduplicator/file2.c
[2025-05-11 23:19:20] [INFO] Завершено, повтор через 20 сек
[2025-05-11 23:19:40] [INFO] Начало обхода
/Users/Roman/Desktop/test-deduplicator
[2025-05-11 23:19:40] [INFO] Завершено, повтор через 20 сек
[2025-05-11 23:20:27] [INFO] SIGTERM получен
```

Для удаления программы необходимо ввести команду:

```
sudo make uninstall
```

Список команд для тестирования программы:

```
mkdir -p ~/Desktop/test-deduplicator
cd ~/Desktop/test-deduplicator

echo "duplicate test" > fileA.txt
cp fileA.txt fileB.txt
cp fileA.txt fileC.txt

cp /Users/Roman/Documents/deduplar/src/deduplar.c file1.c
cp file1.c file2.c
cp file1.c file3.c

deduplar ~/Desktop/test-deduplicator

tail -n +1 -f ~/Library/Caches/deduplar/deduplar.log

ls -li ~/Desktop/test-deduplicator
deduplar -kill
```


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 успешного прохождения всех этапов разработки получено надёжно функционирующее приложение.

Созданная утилита выполняет следующие функции:

- ввод пользователем директории для сканирования;
- запуск фонового процесса для обнаружения дубликатов;
- корректное завершение или принудительное удаление фонового процесса;
- замена найденных дубликатов на жёсткие ссылки;
- логирование результатов фонового процесса;
- постоянный контроль за появлением дублирующихся файлов в файловой системе.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1]. RFC 1321. The MD5 Message-Digest Algorithm - спецификация алгоритма MD5 [Электронный ресурс]. – Режим доступа: <https://www.ietf.org/rfc/rfc1321.txt>.
- [2]. OpenSSL Cookbook– пособие по использованию OpenSSL EVP API [Электронный ресурс]. – Режим доступа: <https://www.feistyduck.com/books/openssl-cookbook/>
- [3]. Хабр – информационный портал для разработчиков [Электронный ресурс]. – Режим доступа: <https://habr.com/ru>.
- [4]. Apple Developer Documentation: Daemons and Services Programming Guid [Электронный ресурс]. – Режим доступа: <https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/Introduction.html>

ПРИЛОЖЕНИЕ А
(Обязательное)

Блок-схема алгоритма функции main

ПРИЛОЖЕНИЕ Б
(Обязательное)

Блок-схема алгоритма функции lookup_duplicate

ПРИЛОЖЕНИЕ В

(Обязательное)

Блок-схема алгоритма функции `compute_md5`

ПРИЛОЖЕНИЕ Г
(Обязательное)

Код программы

```

#define _POSIX_C_SOURCE 200809L
#include <stdio.h> // FILE*, fopen, fprintf, snprintf
#include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE,
malloc, free, getenv, exit
#include <string.h> // strcmp, strlen, snprintf, strdup, memcmp
#include <dirent.h> // DIR, opendir, readdir, closedir
#include <sys/stat.h> // stat, S_ISREG, S_ISDIR
#include <sys/types.h> // off_t, pid_t
#include <openssl/evp.h> // EVP_MD_CTX, EVP_md5, EVP_DigestInit_ex
и др.
#include <unistd.h> // fork, unlink, getpid, sleep, chdir, close,
write
#include <time.h> // time, localtime, strftime
#include <fcntl.h> // open, O_CREAT, O_EXCL, O_WRONLY
#include <syslog.h> // syslog, LOG_ERR
#include <signal.h> // signal, SIGTERM, SIGINT, SIGSEGV, SIGABRT
#include <errno.h> // errno

#include <limits.h> // PATH_MAX

/*
=====
Общие константы
=====
*/
#define APP_NAME "deduplar" // имя утилиты
#define APP_CACHE_FMT "%s/Library/Caches/" APP_NAME // формат
папки кэша
#define HASH_TEXT_LEN 33 // 32 байта hex-MD5 + '\0'
#define MSG_BUF_LEN 700 // размер буфера для сообщений
#define SCAN_INTERVAL 20 // пауза между сканированиями (сек)

/* -----
Глобальные переменные (заполняются в main)
-----
*/
static char g_cache_dir[PATH_MAX]; // каталог для логов, lock-
файла и bin-файла
static char g_log_path [PATH_MAX]; // путь к текстовому логу
static char g_lock_path[PATH_MAX]; // путь к lock-файлу
static char g_bin_path [PATH_MAX]; // путь к бинарному журналу
static int g_lock_fd = -1; // дескриптор lock-файла
(открыт до завершения)

/* -----
Структура, описывающая файл при обходе
-----
*/
typedef struct {
    off_t path_len; // длина пути (включая '\0')
    char *path; // NUL-terminated путь к файлу
    off_t size; // размер файла, байт
    char md5[HASH_TEXT_LEN]; // hex-строка MD5
} file_info_t;

```

```

/* -----
Вспомогательная функция: собрать путь по формату и HOME
-----
*/
static inline void make_path(char *dst, size_t dstsz,
                             const char *fmt, const char *home)
{
    snprintf(dst, dstsz, fmt, home);
}

/* -----
Прототипы всех функций
-----
*/
static void    log_msg(const char *lvl, const char *msg);
static int     same_inode(const char *a, const char *b);
static void    write_bin(const file_info_t *fi);
static char    *lookup_duplicate(const file_info_t *needle);
static void    compute_md5(const char *path, char
out[HASH_TEXT_LEN]);
static void    traverse(const char *dir);
static void    reset_bin(void);
static void    cleanup(void);
static void    on_signal(int sig);
static void    daemonize(void);
static int     ensure_singleton(const char *lock_path);
static int     create_lock_file(const char *lock_path);
static void    kill_running(const char *lock_path);

/*
=====
Точка входа
=====
*/
int main(int argc, char *argv[])
{
    // -----
    // 1. Получаем HOME для формирования путей
    // -----
    const char *home = getenv("HOME");
    if (!home) {
        fprintf(stderr, "%s: HOME не задан\n", APP_NAME);
        return EXIT_FAILURE;
    }

    // -----
    // 2. Формируем пути к cache-каталогу
    // -----
    make_path(g_cache_dir, sizeof(g_cache_dir), APP_CACHE_FMT,
home);
    make_path(g_log_path, sizeof(g_log_path),
APP_CACHE_FMT "/" APP_NAME ".log", home);

```



```

make_path(g_lock_path, sizeof(g_lock_path),
          APP_CACHE_FMT "/" APP_NAME ".lock", home);
make_path(g_bin_path, sizeof(g_bin_path),
          APP_CACHE_FMT "/" APP_NAME ".bin", home);

// -----
// 3. Обработка опции "-kill": если указано, шлём SIGTERM
запущенному демону
// -----
if (argc > 1 && strcmp(argv[1], "-kill") == 0) {
    kill_running(g_lock_path);
    return EXIT_SUCCESS;
}

// -----
// 4. Убедимся, что каталог кэша существует
// -----
char cmd[PATH_MAX + 16];
snprintf(cmd, sizeof(cmd), "mkdir -p \"%s\"", g_cache_dir);
system(cmd);

// -----
// 5. Проверяем, не запущен ли уже другой экземпляр (через
lock-файл)
// -----
if (ensure_singleton(g_lock_path) != 0)
    return EXIT_FAILURE;

// -----
// 6. Устанавливаем обработчики сигналов для корректной
остановки
// -----
signal(SIGTERM, on_signal);
signal(SIGINT, on_signal);
signal(SIGSEGV, on_signal);
signal(SIGABRT, on_signal);

daemonize();

// -----
// 7. Создаём lock-файл (записываем в него PID) и сохраняем FD
открытым
// -----
if ((g_lock_fd = create_lock_file(g_lock_path)) < 0) {
    cleanup();
    return EXIT_FAILURE;
}

// -----
// 8. Выбираем целевую директорию для сканирования (аргумент
или root)
// -----
const char *target = (argc > 1 && argv[1][0] != '-') ? argv[1]
: home;

```

```

// -----
// 9. Бесконечный цикл: логируем, обнуляем журнал, запускаем
обход, спим
// -----
while (1) {
    char buf[MSG_BUF_LEN];

    snprintf(buf, sizeof(buf), "Начало обхода %s", target);
    log_msg("INFO", buf);

    reset_bin(); // очищаем журнал
    traverse(target); // рекурсивно обходим и заменяем
дубликаты

    snprintf(buf, sizeof(buf),
              "Завершено, повтор через %d сек", SCAN_INTERVAL);
    log_msg("INFO", buf);

    sleep(SCAN_INTERVAL);
}

return EXIT_SUCCESS;
}

/*
=====
Реализация функций
=====
*/
/* -----
log_msg: записывает строку в лог-файл с пометкой времени и
уровня
-----
*/
static void log_msg(const char *lvl, const char *msg)
{
    // Открываем лог на дозапись
    FILE *f = fopen(g_log_path, "a");
    if (!f) {
        // При ошибке пишем в syslog и выходим
        syslog(LOG_ERR, "Ошибка открытия %s: %s", g_log_path,
strerror(errno));
        cleanup();
        exit(EXIT_FAILURE);
    }
    // Текущее время
    time_t t = time(NULL);
    char ts[32];
    strftime(ts, sizeof(ts), "%Y-%m-%d %H:%M:%S", localtime(&t));

    // Запись: [YYYY-MM-DD HH:MM:SS] [INFO|ERROR] Сообщение
    fprintf(f, "[%s] [%s] %s\n", ts, lvl, msg);
    fclose(f);
}

```

```

/* -----
   same_inode: проверяет, указывают ли два пути на один и тот же
   файл
   возвращает 1, если inode и устройство совпадают, иначе 0
   -----
*/
static int same_inode(const char *a, const char *b)
{
    struct stat sa, sb;
    if (stat(a, &sa) != 0 || stat(b, &sb) != 0)
        return 0;
    return sa.st_dev == sb.st_dev && sa.st_ino == sb.st_ino;
}

/* -----
   write_bin: добавляет запись о файле в бинарный журнал
   формат записи: path_len + path + size + md5
   -----
*/
static void write_bin(const file_info_t *fi)
{
    FILE *f = fopen(g_bin_path, "ab");
    if (!f) {
        log_msg("ERROR", "Не удалось открыть бинарный журнал");
        cleanup();
        exit(EXIT_FAILURE);
    }
    fwrite(&fi->path_len, sizeof(fi->path_len), 1, f);
    fwrite( fi->path,      1, fi->path_len, f);
    fwrite(&fi->size,      sizeof(fi->size), 1, f);
    fwrite( fi->md5,        1, HASH_TEXT_LEN, f);
    fclose(f);
}

/* -----
   lookup_duplicate: ищет в бинарном журнале первый файл с тем же
   размером и MD5, что needle.
   Возвращает path найденного дубликата (malloc'ит!), или NULL
   -----
*/
static char *lookup_duplicate(const file_info_t *needle)
{
    FILE *f = fopen(g_bin_path, "rb");
    if (!f) return NULL; // журнал пока не создан

    while (1) {
        file_info_t fi;
        if (!fread(&fi.path_len, sizeof(fi.path_len), 1, f))
            break; // EOF

        // читаем путь
        fi.path = malloc(fi.path_len);
        fread(fi.path, 1, fi.path_len, f);
    }
}

```

```

        // читаем размер и md5
        fread(&fi.size,      sizeof(fi.size), 1, f);
        fread( fi.md5,      1, HASH_TEXT_LEN, f);

        // сравниваем
        if (fi.size == needle->size
            && memcmp(fi.md5, needle->md5, HASH_TEXT_LEN) == 0)
        {
            fclose(f);
            return fi.path; // нашли дубликат
        }
        free(fi.path);
    }

    fclose(f);
    return NULL; // не нашли
}

/* -----
   compute_md5: вычисляет MD5-фингерпринт файла в виде hex-строки
   ----- */
static void compute_md5(const char *path, char out[HASH_TEXT_LEN])
{
    // буфер для байтов хеша и длина
    unsigned char hash[EVP_MAX_MD_SIZE];
    unsigned int len = 0;

    // создаём контекст OpenSSL
    EVP_MD_CTX *ctx = EVP_MD_CTX_new();
    FILE *f = fopen(path, "rb");
    if (!ctx || !f) {
        log_msg("ERROR", "Не удалось открыть файл для MD5");
        if (ctx) EVP_MD_CTX_free(ctx);
        if (f) fclose(f);
        return;
    }

    // инициализация MD5
    EVP_DigestInit_ex(ctx, EVP_md5(), NULL);

    // читаем по кускам и обновляем контекст
    char buf[4096];
    size_t r;
    while ((r = fread(buf, 1, sizeof(buf), f)) > 0) {
        EVP_DigestUpdate(ctx, buf, r);
    }

    // финализируем и получаем байты
    EVP_DigestFinal_ex(ctx, hash, &len);

    // очищаем
    EVP_MD_CTX_free(ctx);
    fclose(f);
}

```

```

    // конвертируем байты в hex-строку
    for (unsigned i = 0; i < len; ++i) {
        sprintf(&out[i*2], "%02x", hash[i]);
    }
    out[HASH_TEXT_LEN - 1] = '\\0';
}

/* -----
   traverse: рекурсивно обходит dir, для каждого файла:
   - вычисляет MD5 и ищет дубликат
   - если дубликат найден и не тот же inode:
       удаляет файл и создаёт жесткую ссылку на оригинал
   - иначе записывает в журнал
   -----
*/
static void traverse(const char *dir)
{
    DIR *d = opendir(dir);
    if (!d) {
        log_msg("ERROR", "Не удалось открыть каталог");
        return;
    }
    struct dirent *e;
    while ((e = readdir(d))) {
        // пропускаем . и ..
        if (strcmp(e->d_name, ".") == 0 || strcmp(e->d_name, "..")
== 0)
            continue;

        // формируем полный путь
        char path[PATH_MAX];
        snprintf(path, sizeof(path), "%s/%s", dir, e->d_name);

        struct stat st;
        if (stat(path, &st) != 0) continue;

        if (S_ISDIR(st.st_mode)) {
            // рекурсивно в поддиректорию
            traverse(path);
        }
        else if (S_ISREG(st.st_mode)) {
            // обрабатываем файл
            file_info_t fi;
            fi.size      = st.st_size;
            fi.path_len  = strlen(path) + 1;
            fi.path       = strdup(path);

            compute_md5(path, fi.md5);

            char *dup = lookup_duplicate(&fi);
            if (!dup) {
                // новый уникальный файл — запомним
                write_bin(&fi);
            }
            else if (!same_inode(dup, fi.path)) {

```

```

        // найден дубликат на другом inode'е
        unlink(fi.path);
        if (link(dup, fi.path) == 0) {
            char msg[MSG_BUF_LEN];
            snprintf(msg, sizeof(msg),
                    "Заменено %s -> жесткая ссылка на %s",
                    fi.path, dup);
            log_msg("INFO", msg);
        } else {
            log_msg("ERROR", "Не удалось создать
hardlink");
        }
    }
    free(fi.path);
    if (dup) free(dup);
}
}
closedir(d);
}

/* -----
   reset_bin: сбрасывает бинарный журнал для нового прохода
   -----
*/
static void reset_bin(void)
{
    FILE *f = fopen(g_bin_path, "wb");
    if (f) fclose(f);
}

/* -----
   cleanup: закрывает lock-файл, удаляет lock и bin-файлы
   -----
*/
static void cleanup(void)
{
    if (g_lock_fd >= 0) close(g_lock_fd);
    unlink(g_lock_path);
    unlink(g_bin_path);
}

/* -----
   on_signal: обработчик завершения по сигналам SIGTERM, SIGINT и
   др.
   сохраняет лог, чистит и выходит
   -----
*/
static void on_signal(int sig)
{
    switch (sig) {
        case SIGINT: log_msg("INFO", "SIGINT получен"); break;
        case SIGTERM: log_msg("INFO", "SIGTERM получен"); break;
        case SIGSEGV: log_msg("ERROR", "SIGSEGV - сегфолт"); break;
    }
}

```

```

        case SIGABRT: log_msg("ERROR", "SIGABRT - аварийный
сброс"); break;
        default: break;
    }
    cleanup();
    exit((sig==SIGINT||sig==SIGTERM) ? EXIT_SUCCESS :
EXIT_FAILURE);
}

/* -----
daemonize: делает двойной fork, setsid, перенаправляет
stdin/out/err

.....
1) Первый fork + exit родителя
2) setsid() - новый сеанс, управляющий терминал отрывается
3) Игнорируем SIGHUP
4) Второй fork + exit первого потомка
5) chdir("/") и umask(0)
6) Перенаправление STDIN/STDOUT/STDERR в /dev/null
-----
*/
static void daemonize(void)
{
    pid_t pid = fork();
    if (pid < 0) exit(EXIT_FAILURE);
    if (pid > 0) exit(EXIT_SUCCESS);

    if (setsid() < 0) exit(EXIT_FAILURE);
    signal(SIGHUP, SIG_IGN);

    pid = fork();
    if (pid < 0) exit(EXIT_FAILURE);
    if (pid > 0) exit(EXIT_SUCCESS);

    chdir("/");
    umask(0);

    int fd = open("/dev/null", O_RDWR);
    if (fd >= 0) {
        dup2(fd, STDIN_FILENO);
        dup2(fd, STDOUT_FILENO);
        dup2(fd, STDERR_FILENO);
        if (fd > 2) close(fd);
    }
}

/* -----
ensure_singleton: пробует создать уникальный lock-файл.
если файл уже существует - сообщает, что экземпляр запущен
-----
*/

```

```

static int ensure_singleton(const char *lock_path)
{
    int fd = open(lock_path, O_CREAT|O_EXCL|O_WRONLY, 0644);
    if (fd < 0) {
        if (errno == EEXIST)
            fprintf(stderr, APP_NAME ": уже запущен\n");
        else
            perror(APP_NAME ": lock error");
        return -1;
    }
    close(fd);
    unlink(lock_path);
    return 0;
}

/* -----
   create_lock_file: создаёт lock-файл, записывает в него PID,
   возвращает открытый дескриптор для удержания блокировки
   -----
*/
static int create_lock_file(const char *lock_path)
{
    int fd = open(lock_path, O_WRONLY|O_CREAT|O_TRUNC, 0644);
    if (fd < 0) {
        perror("lock create");
        return -1;
    }
    char buf[32];
    int n = snprintf(buf, sizeof(buf), "%d\n", (int)getpid());
    write(fd, buf, n);
    return fd;
}

/* -----
   kill_running: читает PID из lock-файла и шлёт SIGTERM
   -----
*/
static void kill_running(const char *lock_path)
{
    FILE *f = fopen(lock_path, "r");
    if (!f) {
        fprintf(stderr, APP_NAME ": процесс не найден\n");
        return;
    }
    int pid;
    if (fscanf(f, "%d", &pid) == 1) {
        if (kill(pid, SIGTERM) == 0)
            fprintf(stderr, APP_NAME ": отправлен SIGTERM %d\n",
pid);
        else
            perror(APP_NAME ": kill error");
    }
    fclose(f);
}

```