**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчёт   
ЛАБОРАТОРНАЯ РАБОТА № 1

**по дисциплине «Операционные Системы»**

Тема: **Управление файловой системой**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 |  | Шарпинский Д. А. |
| Преподаватель |  | Тимофеев А. В. |

Санкт-Петербург

2025

# ЦЕЛЬ РАБОТЫ:

Исследование управления файловой системой с помощью Win32 API. Копирование файла с помощью операций перекрывающегося ввода-вывода.

# ЗАДАНИЯ НА ЛАБОРАТОРНУЮ РАБОТУ:

**Задание 1.1. Управление дисками, каталогами и файлами.**

1. Создайте консольное приложение с меню (каждая выполняемая

функция и/или операция должна быть доступна по отдельному пункту меню),

которое выполняет:

∙ вывод списка дисков (функции Win32 API – GetLogicalDrives,

GetLogicalDriveStrings);

∙ для одного из выбранных дисков вывод информации о диске и размер

свободного пространства (функции Win32 API –

GetDriveType, GetVolumeInformation, GetDiskFreeSpace); ∙

создание и удаление заданных каталогов (функции Win32 API –

CreateDirectory, RemoveDirectory);

∙ создание файлов в новых каталогах (функция Win32 API –

CreateFile);

∙ копирование и перемещение файлов между каталогами с возможностью

выявления попытки работы с файлами, имеющими совпадающие

имена (функции Win32 API – CopyFile, MoveFile, MoveFileEx);

∙ анализ и изменение атрибутов файлов (функции Win32 API –

GetFileAttributes, SetFileAttributes, GetFileInformationByHandle,

GetFileTime, SetFileTime).

2. Запустите приложение и проверьте его работоспособность на

нескольких наборах вводимых данных. Запротоколируйте результаты в отчет.

Дайте свои комментарии в отчете относительно выполнения функций Win32

API.

3. Подготовьте итоговый отчет с развернутыми выводами по

заданию.

**Задание 1.2. Копирование файла с помощью операций перекрывающегося ввода-вывода** (будет изменено на выполнение в

Linux).

Приложение должно копировать существующий файл в новый файл,

«одновременно» выполняя n перекрывающихся операций ввода-вывода

(механизм APC) блоками данных кратными размеру кластера.

Указания к выполнению.

1. Создайте консольное приложение, которое выполняет: −

открытие/создание файлов (функция Win32 API – CreateFile,

обязательно использовать флаги FILE\_FLAG\_NO\_BUFFERING и

FILE\_FLAG\_OVERLAPPED);

∙ файловый ввод-вывод (функции Win32 API – ReadFileEx,

WriteFileEx) блоками кратными размеру кластера;

∙ ожидание срабатывания вызова функции завершения (функция Win32

API – SleepEx);

∙ измерение продолжительности выполнения операции копирования файла

(функция Win32 API – TimeGetTime). 2. Запустите приложение и проверьте

его работоспособность на копировании файлов разного размера для ситуации

с перекрывающимся выполнением одной операции ввода и одной операции

вывода (для сравнения файлов используйте консольную команду FC).

Выполните эксперимент для разного размера копируемых блоков, постройте

график зависимости скорости копирования от размера блока данных.

Определите оптимальный размер блока данных, при котором скорость

копирования наибольшая. Запротоколируйте результаты в отчет. Дайте свои

комментарии в отчете относительно выполнения функций Win32 API.

3. Произведите замеры времени выполнения приложения для разного

числа перекрывающихся операций ввода и вывода (1, 2, 4, 8, 12, 16), не

забывая проверять работоспособность приложения (консольная команда FC).

По результатам измерений постройте график зависимости и определите число

перекрывающихся операций ввода и вывода, при котором достигается

наибольшая скорость копирования файла. Запротоколируйте результаты в

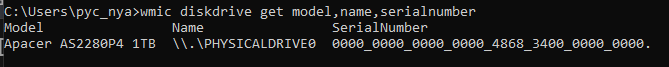
отчет.

4. Подготовьте итоговый отчет с развернутыми выводами по

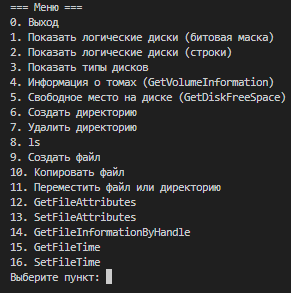
заданию.

# Задание 1.1. Управление дисками, каталогами и файлами.

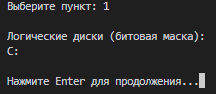
Код запускался на компьютере со следующим SSD:



Главное меню. Демонстрация главного меню функций для работы с файлами:

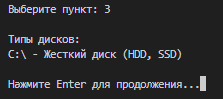


1. Демонстрация работы первой функции главного меню:



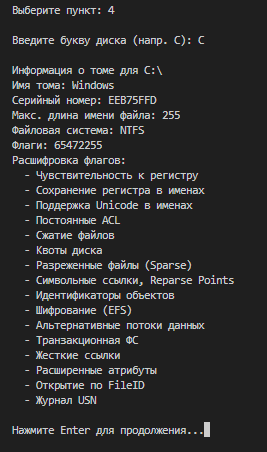
Здесь вызывается showLogicalDrives(), использующая GetLogicalDrives() из WIN32 API.

1. Функция 2 из меню имеет схожий результат работы.
2. Демонстрация работы третьей функции главного меню:

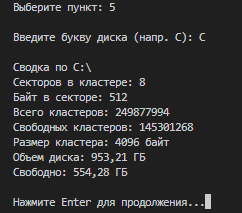


Здесь вызывается showDriveTypes() – выводится тип диска. Использована GetLogicalDriveStrings()

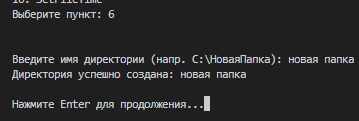
1. Демонстрация работы четвертой функции главного меню:



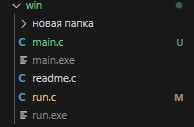
1. Демонстрация работы пятой функции главного меню:



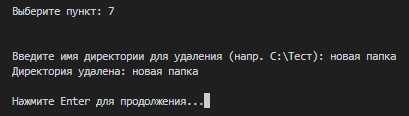
1. Демонстрация работы шестой функции главного меню:



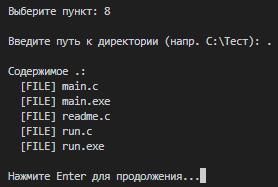
(Была создана новая директория с указанным названием в той же папке, где находился исполняемый файл)



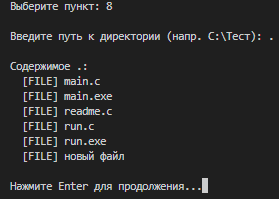
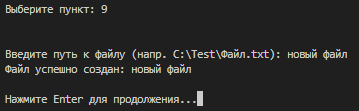
1. Демонстрация работы седьмой функции главного меню:



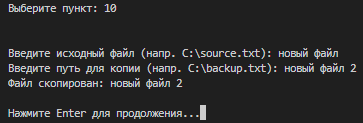
1. Демонстрация работы восьмой функции главного меню:

Данная функция не являлась обязательной к выполнению и была скорее создана для дебага. Результат работы:

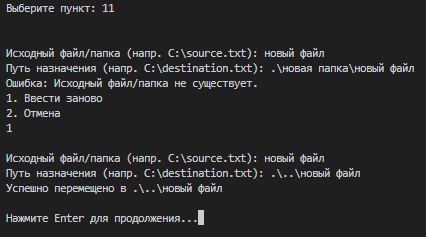
1. Демонстрация работы девятой функции главного меню:



1. Демонстрация работы десятой функции главного меню:

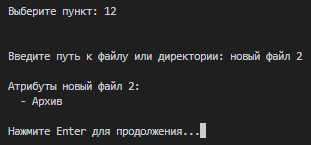


1. Демонстрация работы одиннадцатой функции главного меню:

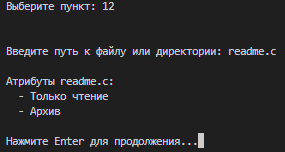
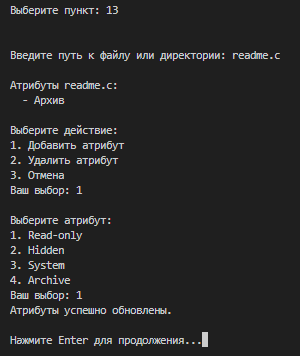


На примере данной функции я бы хотел обратить внимание, что программа устойчива к разным ситуациям и может предложить разные способы решения возможных проблем, как, например, некорректный путь.

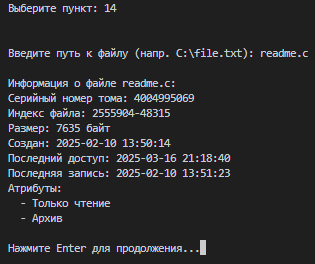
1. Демонстрация работы двенадцатой функции главного меню:



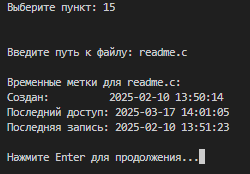
1. Демонстрация работы тринадцатой функции главного меню:



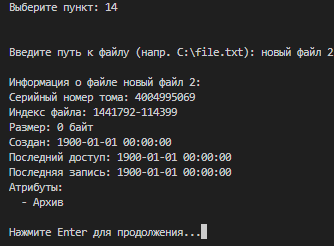
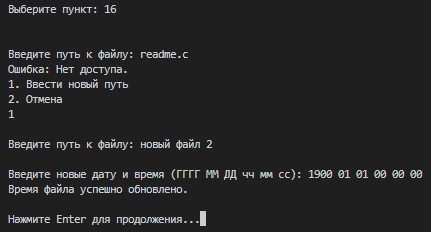
1. Демонстрация работы четырнадцатой функции главного меню:



1. Демонстрация работы пятнадцатой функции главного меню:



1. Демонстрация работы шестнадцатой функции главного меню:



**Итог**

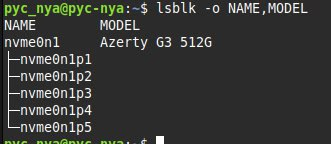
Разработанное консольное приложение успешно решает все пункты задания по работе с дисками, каталогами и файлами в среде Windows. В отчёте приведены примеры тестирования, подтверждающие корректное поведение при разных ситуациях (ошибочных и штатных). Использование Win32 API (CreateFile, CopyFile, MoveFile, GetFileAttributes, SetFileAttributes и проч.) показало, что:

1. Программа способна корректно анализировать состояние файловой системы, работать с атрибутами и атрибутами времени.
2. Обработка ошибок и детальные диалоговые подсказки обеспечивают надёжность и удобство эксплуатации.
3. Все требования задания (от отображения дисков до изменения времени файла) выполнены, что подтверждается экспериментальными проверками.

Данное решение можно считать завершённым и корректным для части 1.1 лабораторной работы.

**Задание 1.2. Копирование файла с помощью операций перекрывающегося ввода-вывода.**

Данная часть работы выполнялась на ноутбуке с операционной системой Linux Mint.



Так как копирование производилось с помощью асинхронных операций необходимо сначала представить реализацию ожидания завершения операций. В данной программе используются два основных механизма синхронизации – **мьютекс** и **условная переменная** – для корректного ожидания завершения асинхронных операций без излишней загрузки процессора (то есть без активного ожидания). Поскольку в программе используется SIGEV\_THREAD, то для каждой асинхронной операции создается новый поток, а значит использование mutex и conditional variable является подходящим решением.

**Мьютекс** – это объект, который гарантирует, что только один поток в определённый момент времени может получить доступ к критической секции кода или разделяемым данным.

**Условная переменная** позволяет потоку (или потокам) «заснуть» и ждать наступления определённого условия (без нагрузки на процессор), вместо того чтобы постоянно проверять состояние (что называется активным ожиданием).

У условных переменных есть механизм notify\_all(), который уведомляет о завершении работы данного потока. При получении уведомления вызывается callback, который проверяет, каждый ли поток завершил работу – если так, то копирование считается завершенным, иначе выполнение продолжается.

* Тестирование проводилось на файлах размерами 10mb, 100mb и 1gb. Файлы генерировались случайно с помощью следующей команды:

dd if=/dev/urandom of=random\_file.bin bs=1M count=10

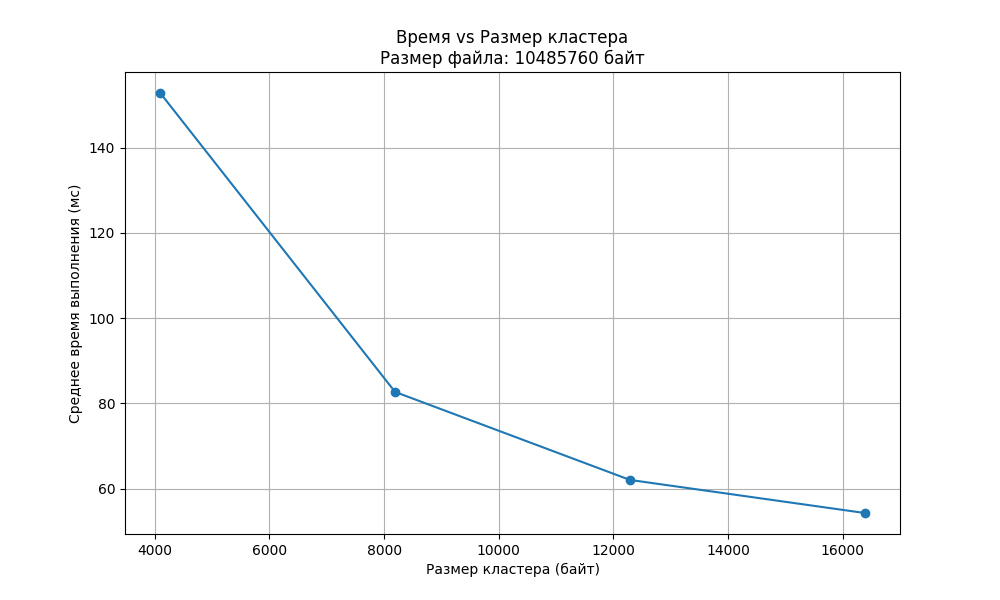
* Для каждого файла проводилось по 5 тестов с одинаковыми входными данными, а именно: количеством байт, отводимых под чтение и запись, и количество одновременных асинхронных операций.
* Между итерациями делался вызов fsync() и sync() для чистоты бенчмарка.
* На начальных этапах была проведена проверка точности копирования с помощью команды:

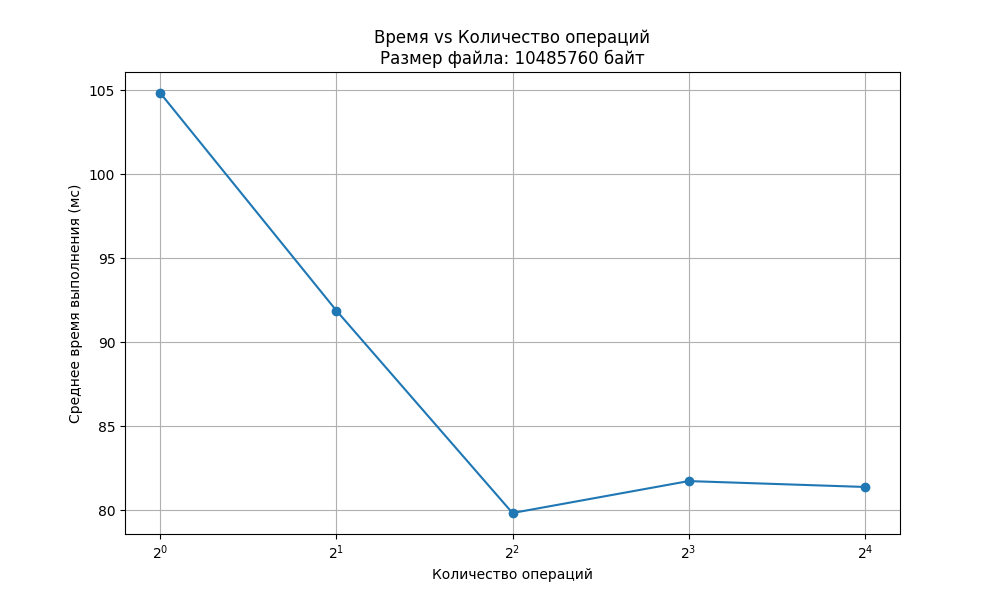
md5sum <file\_name>

Для оригинального файла и его копии. Хэш-сумма совпадала в 100% случаев.

Результаты бенчмарка продемонстрированы на графиках (получены на основании логов программы):

Файл размером 10mb:

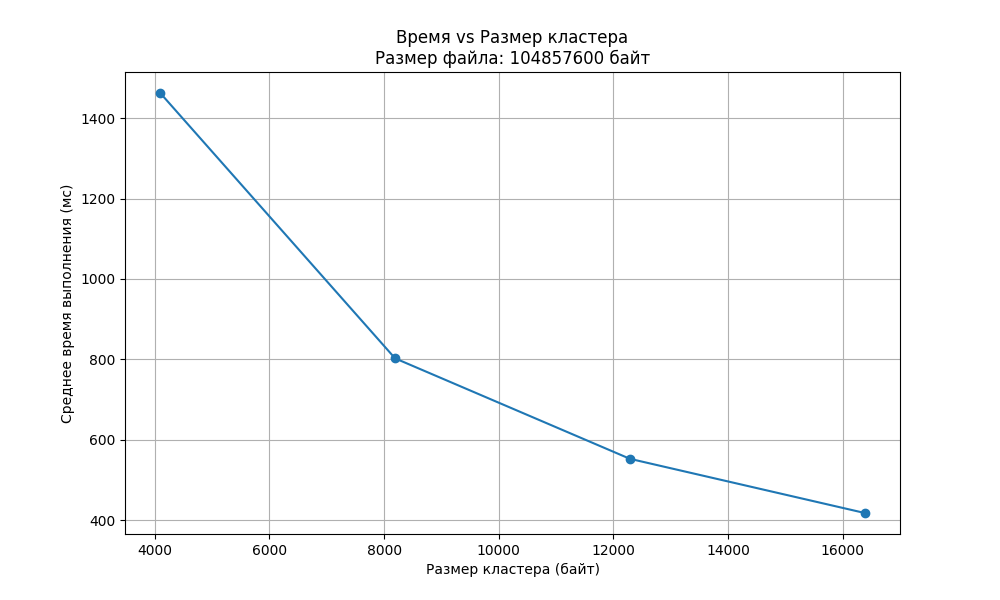


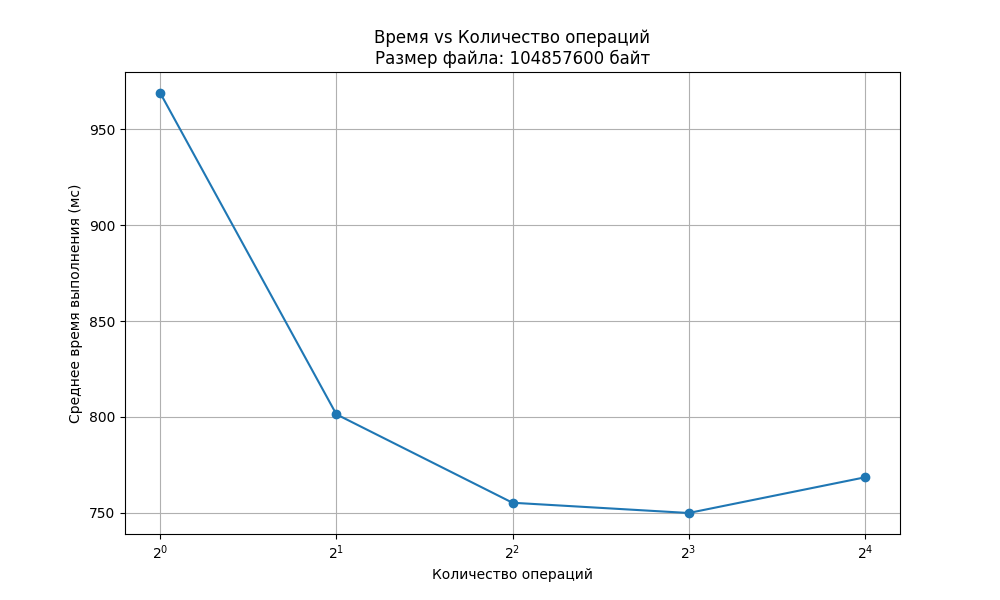


Промежуточный итог:

Для файла размером 10 мегабайт очевидно, что большее количество байт на одну операцию ведет к экспоненциальному понижению времени на копирование. Самыми эффективными показали себя запуски программы с 4 одновременными асинхронными операциями.

Файл размером 100mb:

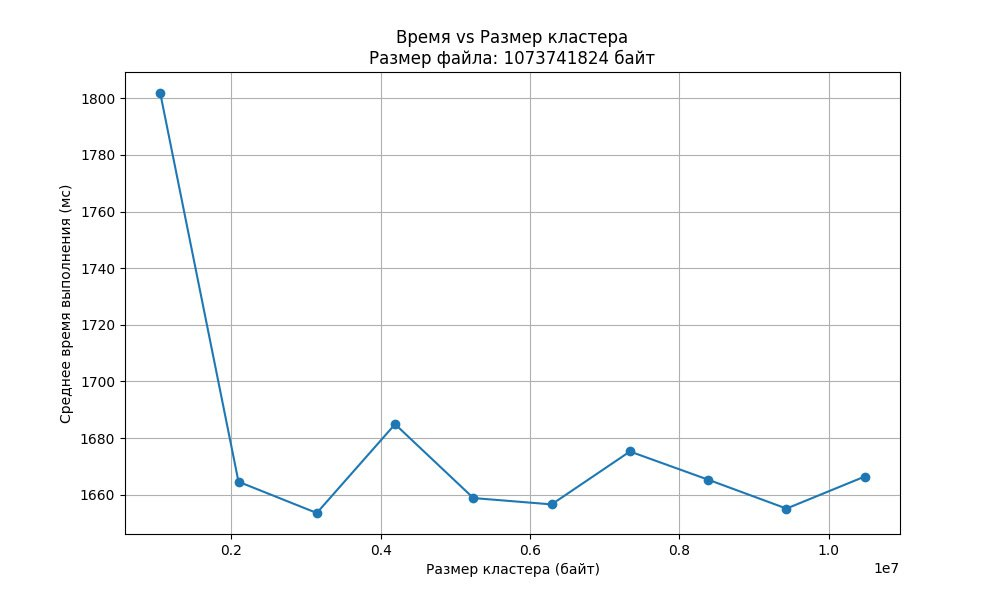


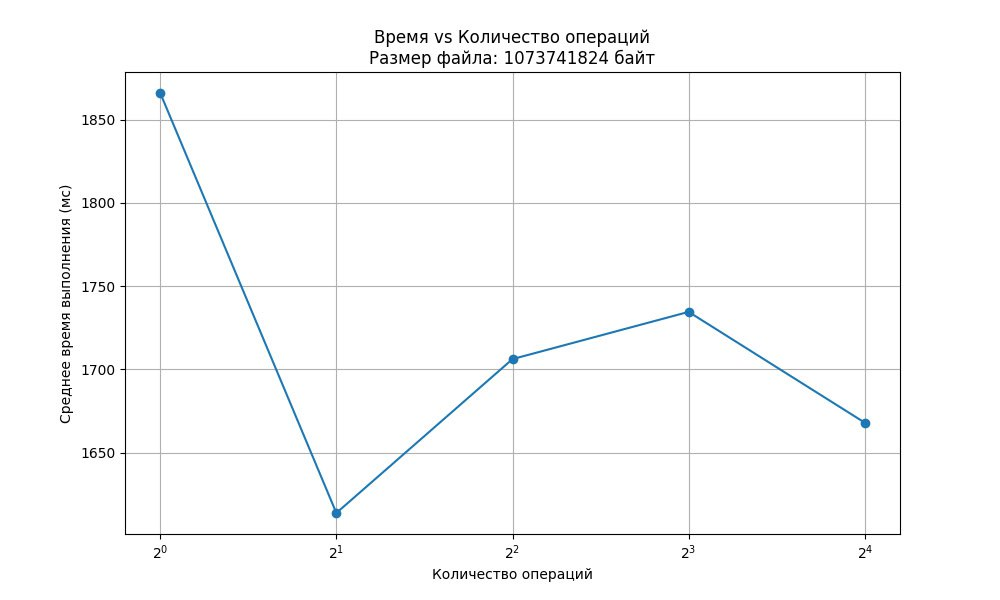


Промежуточный итог:

Для файла размером 100 мегабайт результаты в общем схожи с результатами для файла размером 10мб, но с той разницей, что 8 потоков показали себя более эффективными. Однако разница крайне невелика, так что можно списать это на погрешность.

Файл размером 1gb:





Для проведения бенчмарка для файла такого размера в коде между итерациями был добавлен вызов функции sleep(2), для того, чтобы на 100% гарантировать чистоту эксперимента и независимость каждой итерации.

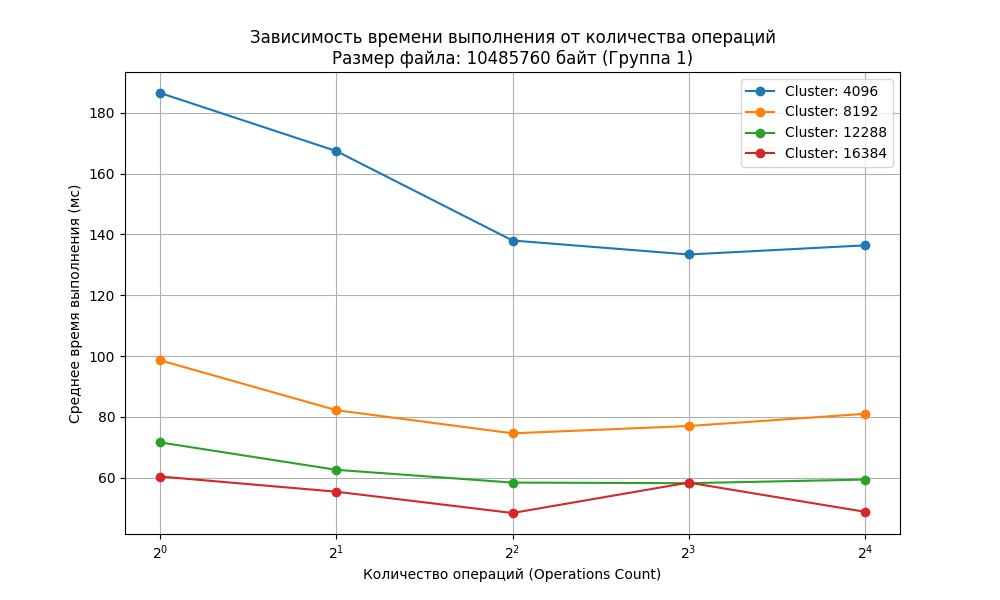
Промежуточный итог:

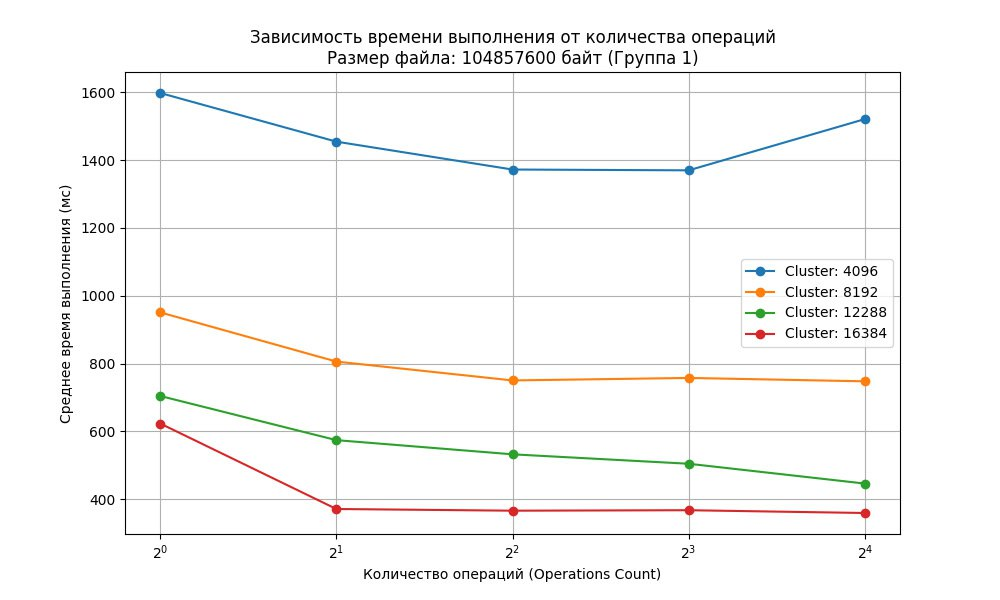
Для файла размером 1 гигабайт результаты в целом сохраняют тенденцию, заданную двумя предыдущими: большему размеру кластера соответствует меньшее время, при большом количестве потоков время может возрастать.

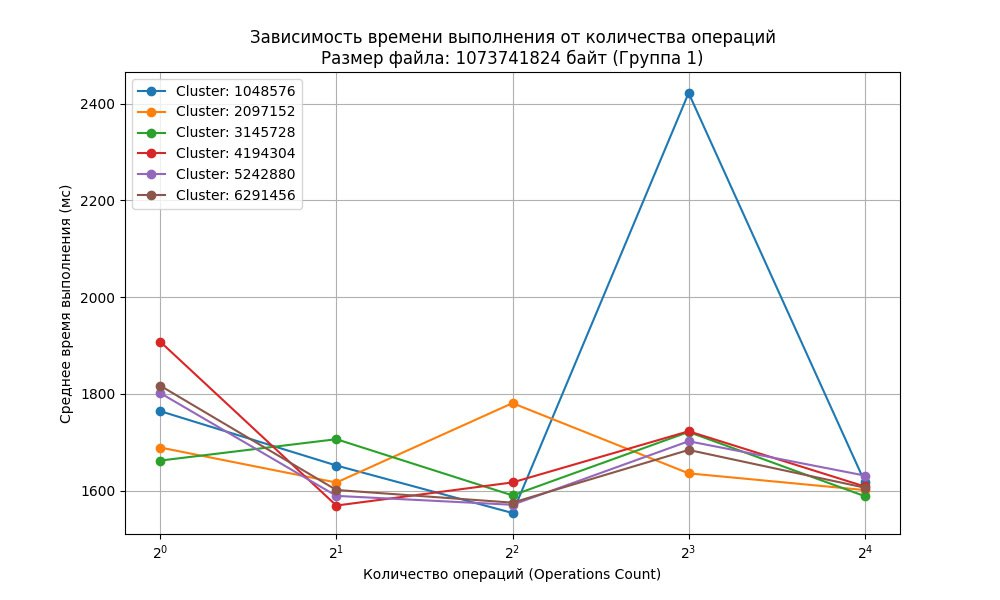
Здесь график кластером выглядит несколько ломанным – это можно объяснить погрешностью измерений, т к разница колеблется в пределах 20мс, что для файла такого размера несущественно.

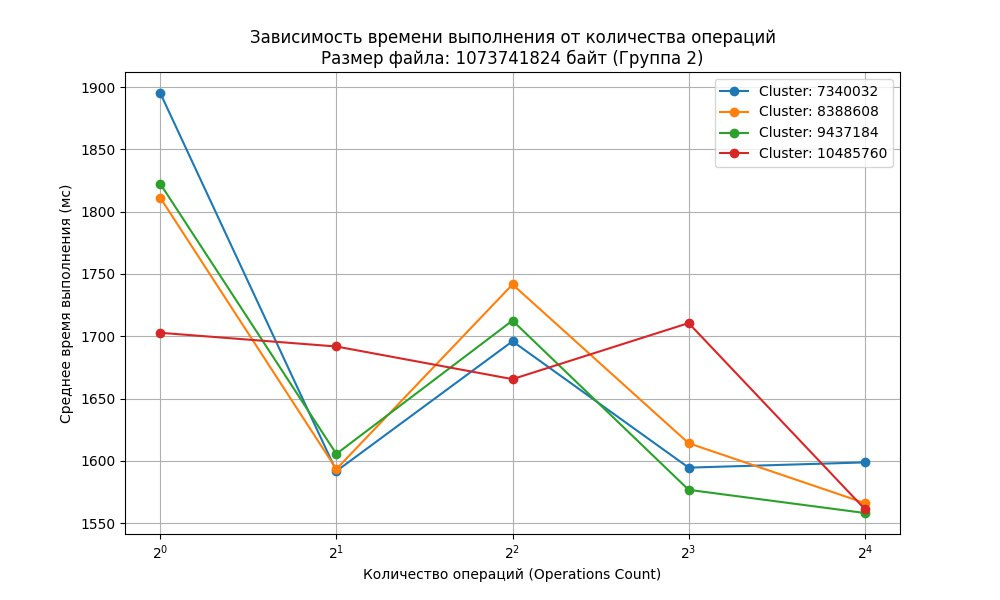
Важно понимать, что для файла такого размера, погрешность при каждой итерации увеличивается.

Также по полученным логам можно было построить несколько иное отображение результатов:









По графикам можно понять, что наибольшая эффективность работы программы для большинства итераций была при 4 потоках.