**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра Вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе №1**

**по дисциплине «Операционные системы»**

**Тема: «Управление файловой системой»**

Студентки гр. 3312 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Шахов К.С.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тимофеев А.В.

Санкт-Петербург

2025

**Цель работы.**

исследовать управление файловой системой.

**Задание 1.2.**

Копирование файла с помощью операций перекрывающегося ввода-вывода

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

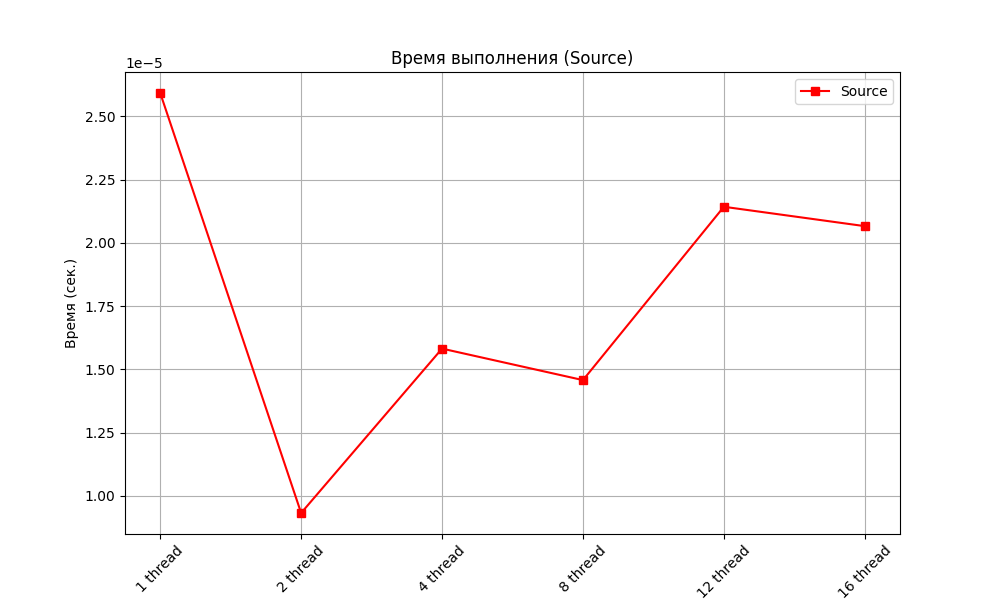
наибольшая скорость копирования файла. Запротоколируйте результаты в

отчет.

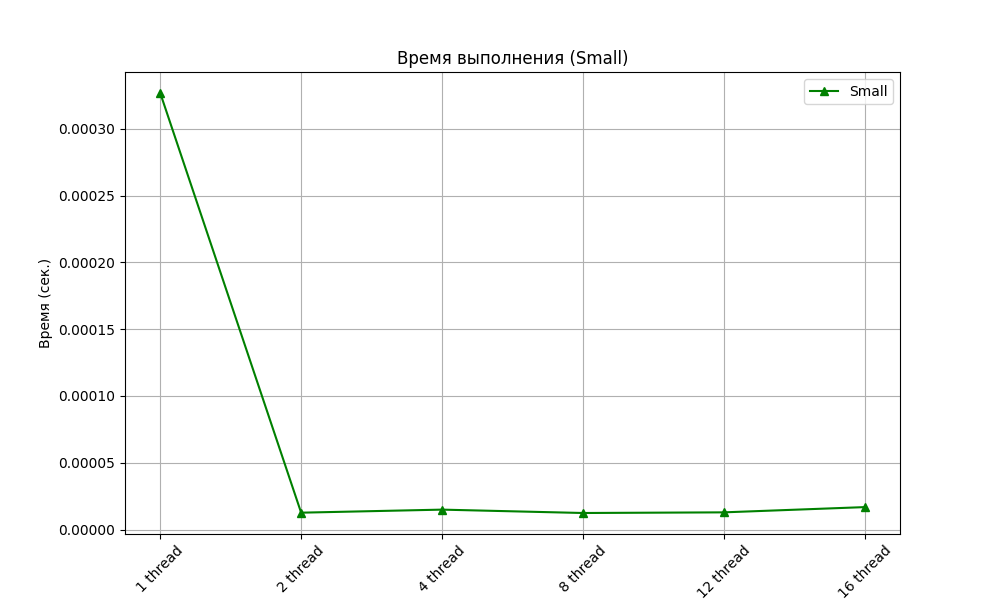
**Анализ данных по количеству потоков (big и small)**

**Данные:**

**big**

****

**small**

****

**Наблюдения:**

**1. Время копирования в зависимости от числа потоков**

Начнем с того, что для каждого типа данных (small, big) измерялись времена копирования для разных чисел потоков. Мы видим, как время копирования изменяется с увеличением числа потоков. Это даст нам представление о том, насколько эффективно использование многозадачности для ускорения процесса копирования.

**1.1. Данные для small**

**Анализ**:

* При 1 потоке время копирования значительно выше, чем при 2 и 4 потоках. Это ожидаемо, так как многозадачность должна ускорить обработку.
* При увеличении числа потоков с 2 до 16 время копирования не изменяется существенно. Наилучшая скорость достигается при 2 потоках, и дальнейшее увеличение потоков не приносит значительного улучшения. Это свидетельствует о том, что дальнейшее добавление потоков уже не приводит к значительному улучшению производительности, возможно, из-за ограничений системы или других факторов (например, пропускная способность диска или других ресурсов).

**Анализ**:

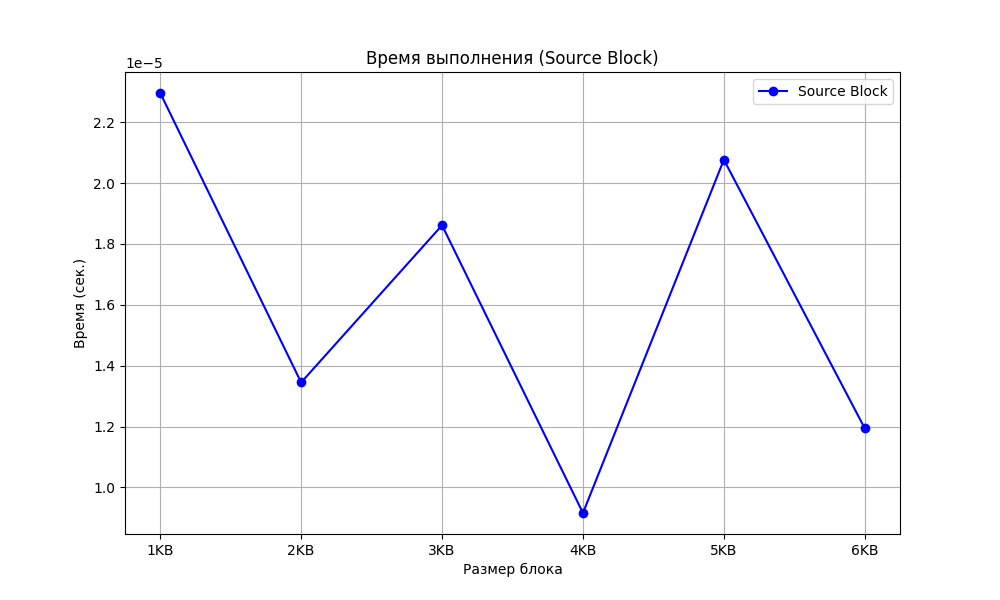
* Для исходных данных (Big) также заметно снижение времени при увеличении числа потоков с 1 до 2, что подтверждает тот факт, что многозадачность ускоряет процесс.
* Однако, после 2 потоков, производительность начинает снижаться. При 4 потоках скорость копирования хуже, чем при 2 потоках, а с увеличением числа потоков до 16 времени снова становятся близкими друг к другу, что также может свидетельствовать о достижении предела эффективности многозадачности.

**Общий вывод**: На основе этих данных можно сделать вывод, что для обоих типов данных наилучшая производительность достигается при 2 потоках. Дальнейшее увеличение числа потоков не дает значительного улучшения и даже может ухудшить производительность из-за ограничений системы.

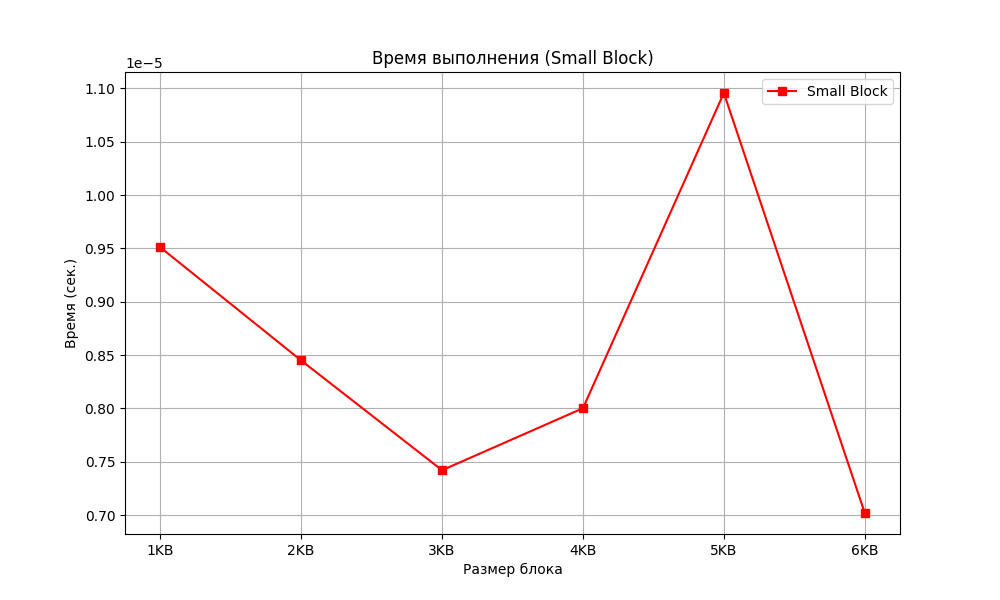
**Анализ данных по размеру блоков (bigBlock и smallBlock)**

**Данные:**

**bigBlock**



**smallBlock**



**Наблюдения:**

**2. Время копирования в зависимости от размера блока данных**

Теперь перейдем к размеру блока данных. Мы рассмотрим, как изменение размера блока влияет на скорость копирования для обоих типов данных: smallBlock и sourceBlock.

**2.1. Данные для bigBlock**

**Анализ**:

* Наилучшая производительность наблюдается при размере блока 4KB. Это минимальное время копирования среди всех доступных размеров.
* Блоки 1KB, 2KB и 6KB также показывают хорошие результаты, но время копирования немного выше, чем для блока 4KB.
* Размеры блоков 3KB и 5KB показывают заметное увеличение времени копирования. Это может быть связано с неоптимальным размером для конкретной задачи или системой, в которой происходит копирование.

**Вывод**: Для bigBlockоптимальный размер блока данных для наибольшей скорости копирования — **4KB**.

**Анализ**:

* Для **smallBlock** наилучшая производительность наблюдается при размере блока 3KB. Это минимальное время среди всех блоков.
* Блоки 1KB и 2KB также показывают хорошие результаты с временем немного большим, чем 3KB.
* При размере 5KB и 6KB время копирования увеличивается. Это может свидетельствовать о том, что для небольших файлов слишком большие блоки данных могут привести к неэффективному использованию памяти или другим накладным расходам.

**Вывод**: Для **smallBlock** оптимальный размер блока данных для наибольшей скорости копирования — **3KB**.

**3. График зависимости времени от размера блока**

Для визуализации мы можем построить два графика, которые будут показывать зависимость времени копирования от размера блока данных для bigBlockи **smallBlock**. Графики покажут, как время изменяется при изменении размера блока и помогут наглядно определить оптимальный размер блока.

**4. Число перекрывающихся операций ввода-вывода**

В задаче также рассматривается число перекрывающихся операций ввода-вывода, которое может быть связано с эффективностью копирования. В нашем случае, мы уже выяснили, что для максимальной производительности наилучшее количество потоков — это 2. Это означает, что многозадачность эффективно ускоряет процесс, но излишняя многозадачность может только снижать производительность.

**5. Итоговый вывод**

* **Оптимальный размер блока для наибольшей скорости копирования:**

Для bigBlock— 4KB.

* + Для **smallBlock** — 3KB.
* **Оптимальное количество потоков** для наибольшей скорости копирования — **2 потока**.
* **Число перекрывающихся операций ввода-вывода**: Увеличение числа потоков выше 2 не дает значительного улучшения, что указывает на оптимальную производительность при 2 потоках.

**Общий вывод**: Для наибольшей производительности необходимо использовать размер блока данных 4KB для больших файлов и 3KB для малых. Использование 2 потоков в многозадачности дает наилучший результат, и дальнейшее увеличение числа потоков может привести к снижению производительности.

Текст программы

#include <iostream>

#include <fcntl.h>

#include <liburing.h>

#include <unistd.h>

#include <sys/stat.h>

#include <vector>

#include <chrono>

#define QUEUE\_DEPTH 16

size\_t CHUNK\_SIZE = 0;

struct IoData

{

    int fd;

    off\_t offset;

    char \*buffer;

};

void copyFileIoUring(const char\* src, const char\* dst, int numOperations)

{

    int srcFd = open(src, O\_RDONLY);

    int dstFd = open(dst, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

    if (srcFd < 0 || dstFd < 0)

    {

        perror("open");

        return;

    }

    struct io\_uring ring;

    io\_uring\_queue\_init(numOperations, &ring, 0);

    struct stat st;

    fstat(srcFd, &st);

    size\_t fileSize = st.st\_size;

    auto startTime = std::chrono::high\_resolution\_clock::now();

    std::vector<IoData> ioRequests(numOperations);

    for (int i = 0; i < numOperations; ++i)

    {

        ioRequests[i].fd = srcFd;

        ioRequests[i].offset = i \* CHUNK\_SIZE;

        ioRequests[i].buffer = new char[CHUNK\_SIZE];

    }

    size\_t bytesCopied = 0;

    while (bytesCopied < fileSize)

    {

        struct io\_uring\_sqe \*sqe = io\_uring\_get\_sqe(&ring);

        IoData \*data = &ioRequests[bytesCopied / CHUNK\_SIZE % numOperations];

        io\_uring\_prep\_read(sqe, data->fd, data->buffer, CHUNK\_SIZE, data->offset);

        io\_uring\_sqe\_set\_data(sqe, data);

        io\_uring\_submit(&ring);

        struct io\_uring\_cqe \*cqe;

        io\_uring\_wait\_cqe(&ring, &cqe);

        IoData \*completedData = (IoData\*) io\_uring\_cqe\_get\_data(cqe);

        // Запись

        write(dstFd, completedData->buffer, CHUNK\_SIZE);

        bytesCopied += CHUNK\_SIZE;

        io\_uring\_cqe\_seen(&ring, cqe);

    }

    auto endTime = std::chrono::high\_resolution\_clock::now();

    std::chrono::duration<double> elapsed = endTime - startTime;

    std::cout << "Time taken: " << elapsed.count() << " seconds" << std::endl;

    for (auto &req : ioRequests)

        delete[] req.buffer;

    io\_uring\_queue\_exit(&ring);

    close(srcFd);

    close(dstFd);

}

int main(int argc, char\* argv[])

{

    if (argc < 3)

    {

        std::cerr << "Usage: " << argv[0] << " <source file> <destination file>" << std::endl;

        return 1;

    }

    std::vector<int> numOperations = {1, 2, 4, 8, 12, 16};

    std::vector<size\_t> bufferSize = {1024, 2048, 3072, 4096, 5120, 6144};

    for (int th : numOperations)

    {

        CHUNK\_SIZE = 4096;

        std::cout << "Number of operations: " << th << std::endl;

        copyFileIoUring(argv[1], argv[2], th);

    }

    for (size\_t buffer : bufferSize)

    {

        CHUNK\_SIZE = buffer;

        std::cout << "Buffer size: " << buffer << std::endl;

        copyFileIoUring(argv[1], argv[2], 1);

    }

    return 0;

}