**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»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 | Баймухамедов Р. Р. |  |
| Преподаватель | Тимофеев А. В. |  |

Санкт-Петербург

2024

**Цель работы**

Приложение должно копировать существующий файл в новый файл,

«одновременно» выполняя n перекрывающихся операций ввода-вывода

(механизм APC) блоками данных кратными размеру кластера.

**Задание**

Создайте консольное приложение, которое выполняет: − открытие/создание файлов. Измерьте продолжительности выполнения операции копирования файла. Проверьте его работоспособность на копировании файлов разного размера для ситуации с перекрывающимся выполнением одной операции ввода и одной операции вывода. Определите оптимальный размер блока данных, при котором скорость копирования наибольшая. Произведите замеры времени выполнения приложения для разного числа перекрывающихся операций ввода и вывода. По результатам измерений постройте график зависимости и определите число перекрывающихся операций ввода и вывода, при котором достигается наибольшая скорость копирования файла.

**Постановка задачи и описание решения**

Для выполнения данной лабораторной работы необходимо разработать консольное приложение, которое

* Размер блока данных – определить, как размер копируемого блока влияет на скорость копирования.
* Число перекрывающихся операций ввода-вывода – найти оптимальное количество одновременных операций, при котором достигается максимальная скорость копирования.

В рамках эксперимента необходимо:

* Провести копирование файлов разного размера при фиксированном числе перекрывающихся и построить график зависимости скорости копирования от размера блока данных.
* Провести копирование файлов при разном числе перекрывающихся операций (1, 2, 4, 8, 12, 16) и построить график зависимости скорости копирования от их количества.
* Определить оптимальные параметры копирования, при которых достигается наибольшая скорость.

**Примеры выполнения программы**

Примеры работоспособности консольного приложения продемонстрированы ниже

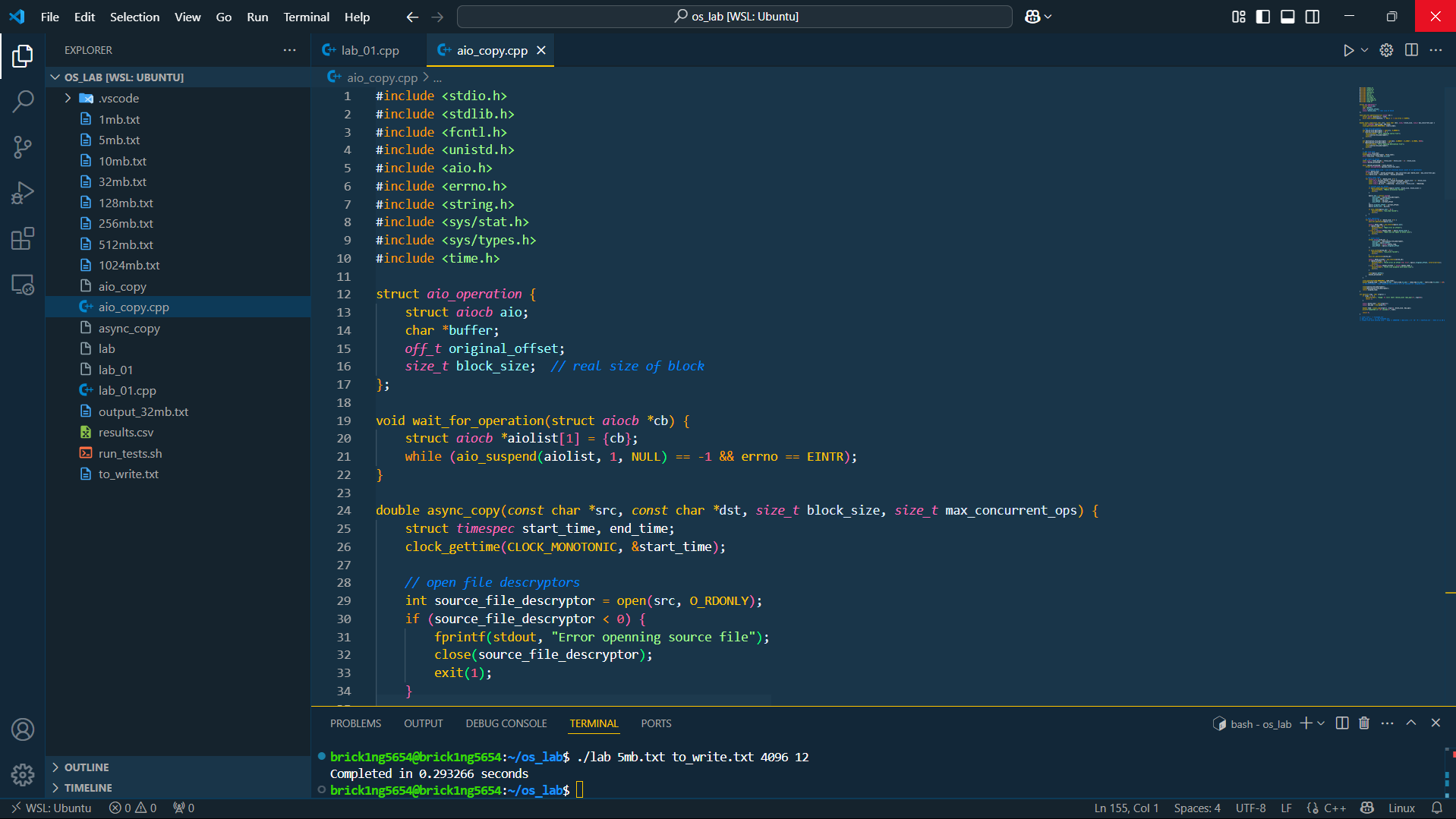
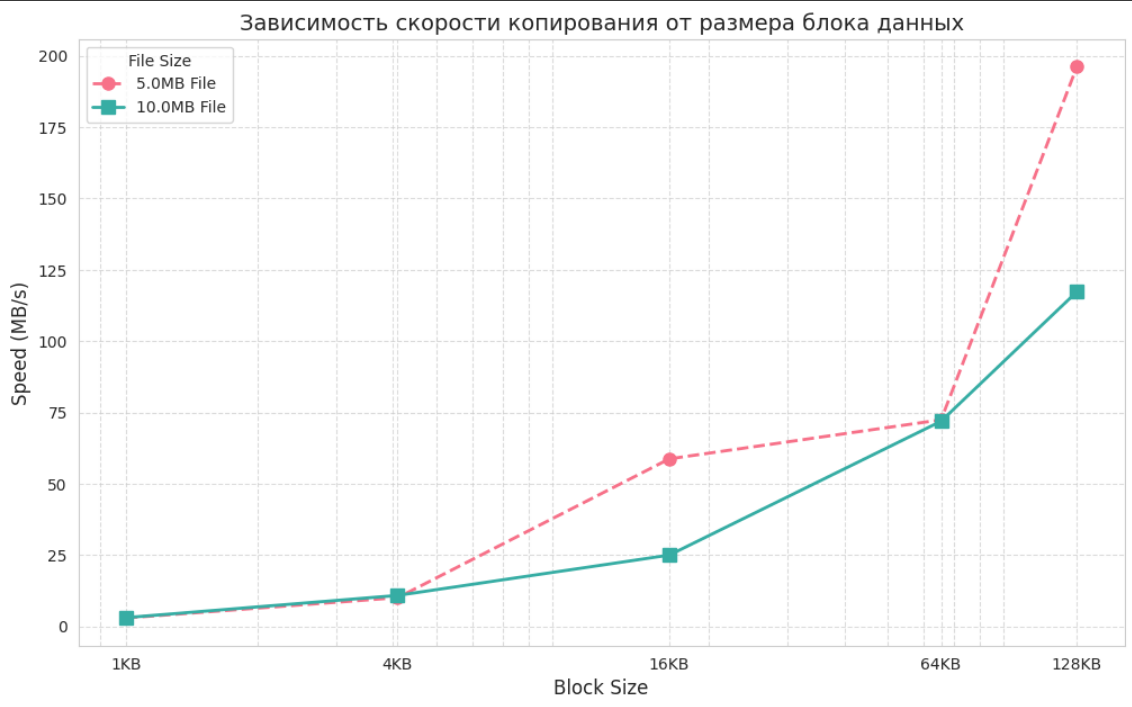
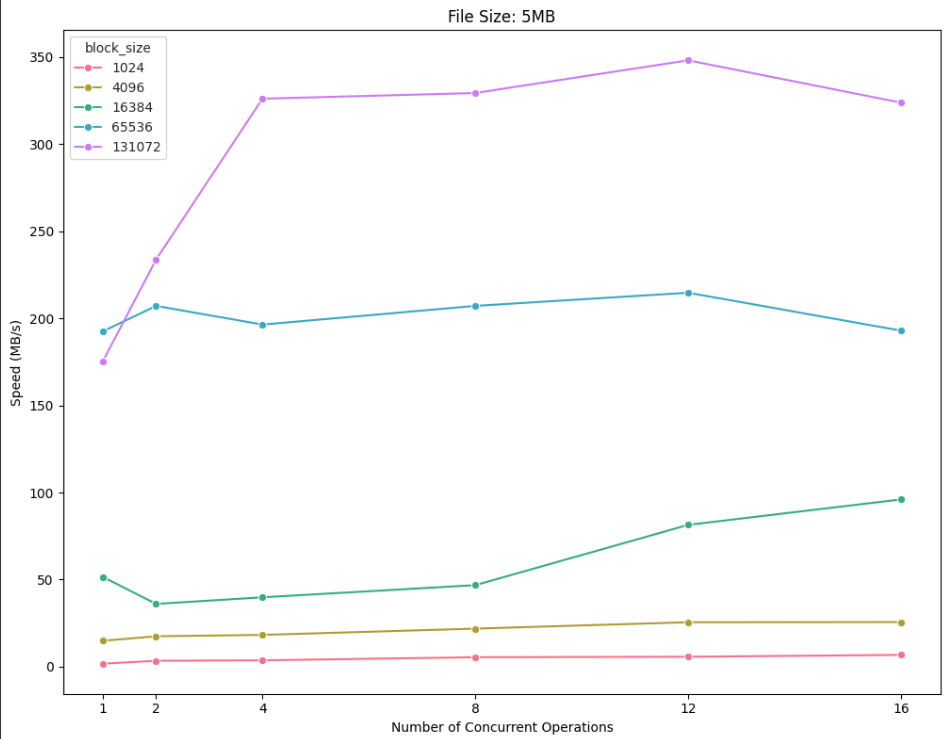
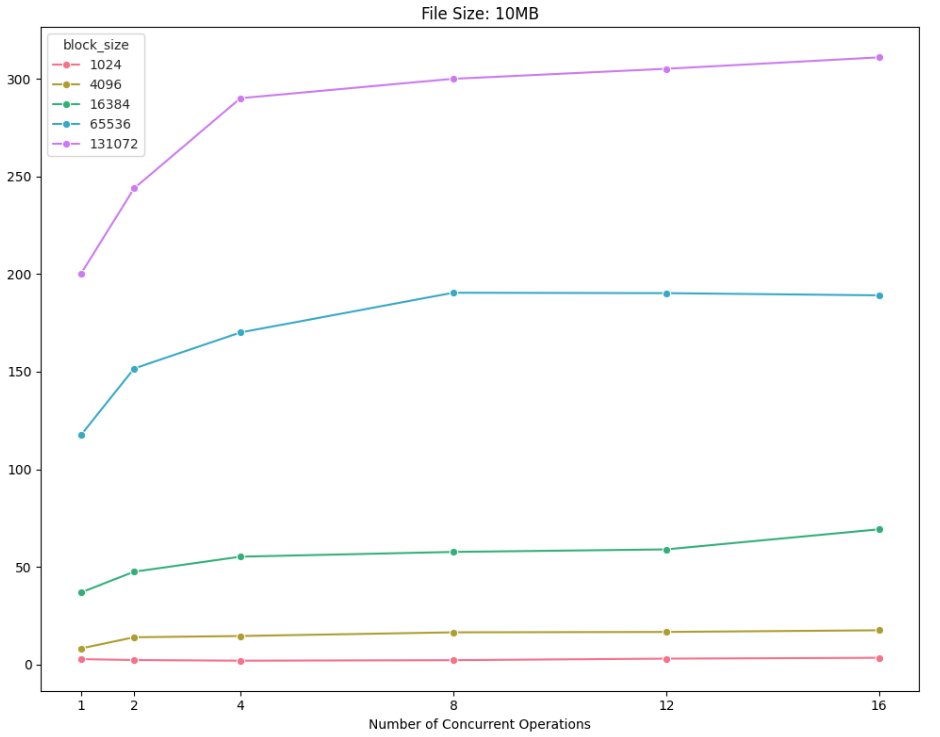


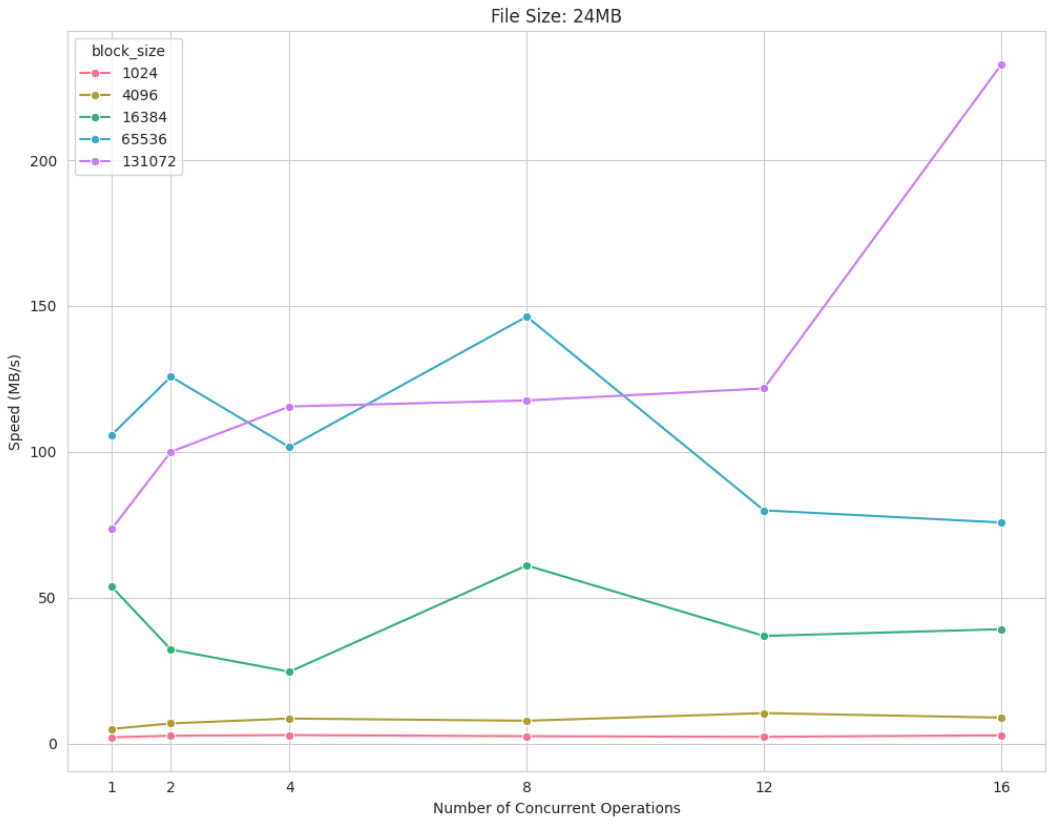
График зависимости скорости копирования от размера блока данных:



Графики зависимости числа перекрывающихся операций ввода и вывода, при котором достигается наибольшая скорость копирования файла:







**Заключение**

**Текст программы**

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <aio.h>

#include <errno.h>

#include <string.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <time.h>

struct *aio\_operation* {

    struct *aiocb* aio;

    char \*buffer;

*off\_t* original\_offset;

*size\_t* block\_size; *// real size of block*

};

void wait\_for\_operation(struct *aiocb* *\**cb) {

    struct *aiocb* *\**aiolist[1] = {cb};

    while (aio\_suspend(aiolist, 1, NULL) == -1 && errno == EINTR);

}

double async\_copy(*const* char *\**src, *const* char *\**dst, *size\_t* block\_size, *size\_t* max\_concurrent\_ops) {

    struct *timespec* start\_time, end\_time;

    clock\_gettime(CLOCK\_MONOTONIC, &start\_time);

*// open file descryptors*

    int source\_file\_descryptor = open(src, O\_RDONLY);

    if (source\_file\_descryptor < 0) {

        fprintf(stdout, "Error openning source file");

        close(source\_file\_descryptor);

        exit(1);

    }

    int destination\_file\_descryptor = open(dst, O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

    if (destination\_file\_descryptor < 0) {

        fprintf(stdout, "Error openning destination file");

        close(source\_file\_descryptor);

        exit(1);

    }

*// get size file*

    struct *stat* file\_stat;

    fstat(source\_file\_descryptor, &file\_stat);

*off\_t* file\_size = file\_stat.st\_size;

*// get count of blocks*

*const* *size\_t* total\_blocks = (file\_size + block\_size - 1) / block\_size;

*size\_t* blocks\_processed = 0;

    while (blocks\_processed < total\_blocks) {

        struct *aio\_operation* ops[max\_concurrent\_ops];

*// calculate batch\_size - size of iteration block (count of io operations)*

*size\_t* batch\_size;

        if ((total\_blocks - blocks\_processed) > max\_concurrent\_ops) batch\_size = max\_concurrent\_ops;

        else batch\_size = total\_blocks - blocks\_processed;

*// start async read*

        for (*size\_t* i = 0; i < batch\_size; i++) {

*const* *off\_t* current\_offset = blocks\_processed \* block\_size + i \* block\_size;

*const* *size\_t* remaining = file\_size - current\_offset;

*const* *size\_t* op\_size = (remaining > block\_size) ? block\_size : remaining;

*// alocate size for buffer*

            if (posix\_memalign((void\*\*)&ops[i].buffer, block\_size, block\_size)) {

                fprintf(stdout, "Memory allocation failed");

                exit(1);

            }

            ops[i].aio = (struct *aiocb*){

                .aio\_fildes = source\_file\_descryptor,

                .aio\_buf = ops[i].buffer,

                .aio\_nbytes = op\_size,

                .aio\_offset = current\_offset

            };

            ops[i].original\_offset = current\_offset;

            ops[i].block\_size = op\_size;

            if (aio\_read(&ops[i].aio) < 0) {

                fprintf(stdout, "aio\_read failed");

                exit(1);

            }

        }

*// wait operations*

        for (*size\_t* i = 0; i < batch\_size; i++) {

            wait\_for\_operation(&ops[i].aio);

*ssize\_t* bytes\_read = aio\_return(&ops[i].aio);

            if (bytes\_read < 0) {

                fprintf(stdout, "Read error at offset");

                exit(1);

            } else if ((*size\_t*)bytes\_read != ops[i].block\_size) {

                fprintf(stdout, "Read size not equal to block size");

                exit(1);

            }

*// async write*

            struct *aiocb* write\_cb = {

                .aio\_fildes = destination\_file\_descryptor,

                .aio\_buf = ops[i].buffer,

                .aio\_nbytes = (*size\_t*)bytes\_read,

                .aio\_offset = ops[i].original\_offset

            };

            if (aio\_write(&write\_cb) < 0) {

                fprintf(stdout, "aio\_write failed");

                exit(1);

            }

            wait\_for\_operation(&write\_cb);

*ssize\_t* bytes\_written = aio\_return(&write\_cb);

            if (bytes\_written < 0) {

                fprintf(stderr, "Write error at offset %ld: %s\n", ops[i].original\_offset, strerror(errno));

                exit(1);

            } else if ((*size\_t*)bytes\_written != (*size\_t*)bytes\_read) {

                fprintf(stderr, "Write size mismatch with block size");

                exit(1);

            }

            free(ops[i].buffer);

            blocks\_processed++;

        }

    }

    clock\_gettime(CLOCK\_MONOTONIC, &end\_time);

    double elapsed\_time = (end\_time.tv\_sec - start\_time.tv\_sec) + (end\_time.tv\_nsec - start\_time.tv\_nsec) / 1e9;

*// fprintf(stdout, "Copy completed successfully in %.6f seconds!\n", elapsed\_time);*

    close(source\_file\_descryptor);

    close(destination\_file\_descryptor);

    return elapsed\_time;

}

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

    if (argc != 5) {

        fprintf(stderr, "Usage: %s <src> <dst> <block\_size> <max\_ops>\n", argv[0]);

        return 1;

    }

*size\_t* block\_size = atoi(argv[3]);

*size\_t* max\_ops = atoi(argv[4]);

    double time = async\_copy(argv[1], argv[2], block\_size, max\_ops);

    printf("Completed in %.6f seconds\n", time);

    return 0;

}

*// clear file - `> filename.txt`*

*// get file size - `ls -lh filename.txt`*

*// create file with entered size - `head -c 10485760 < /dev/zero | tr '\0' '0' > testfile.txt` - here it is 10 mbytes*