МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники

ОТЧЕТ

по лабораторной работе № 1.2

по дисциплине «Операционные системы»

Тема: «Управление файловой системой»

Студент гр. 3311

Пасечный
Л.В.

Преподаватель

Тимофеев
А. В.

Санкт-Петербург
2024

Введение

Цель работы:

исследовать управление файловой системой с
помощью Win32 API.

Постановка задачи:

1. Создайте консольное приложение, которое выполняет: −
открытие/создание файлов
· измерение продолжительности выполнения операции копирования файла.
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.
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.

Результаты:

Block Size (bytes)	Concurrent Ops	Time (seconds)	Speed (MB/s)
1024	1	0.261580	9.12
1024	2	0.121954	19.56
1024	4	0.083060	28.71
1024	8	0.122878	19.41
1024	12	0.114177	20.89
1024	16	0.106604	22.37
4096	1	0.059263	40.25
4096	2	0.031986	74.57
4096	4	0.029773	80.11
4096	8	0.034555	69.02
4096	12	0.031018	76.89
4096	16	0.029176	81.75
8192	1	0.031174	76.51
8192	2	0.020183	118.17
8192	4	0.021675	110.04
8192	8	0.016018	148.90
8192	12	0.016587	143.79
8192	16	0.023407	101.90
16384	1	0.018103	131.75
16384	2	0.018550	128.57
16384	4	0.015726	151.66
16384	8	0.007125	334.75
16384	12	0.010886	219.09
16384	16	0.006310	377.98
32768	1	0.009840	242.38
32768	2	0.020801	114.66
32768	4	0.004133	577.08
32768	8	0.004081	584.43
32768	12	0.012441	191.71
32768	16	0.008499	280.63
65536	1	0.003551	671.66
65536	2	0.003415	698.41
65536	4	0.003076	775.38
65536	8	0.003045	783.27
65536	12	0.002986	798.75
65536	16	0.003319	718.61
131072	1	0.008544	279.15
131072	2	0.007668	311.04

График график зависимости скорости копирования от размера блока данных.:

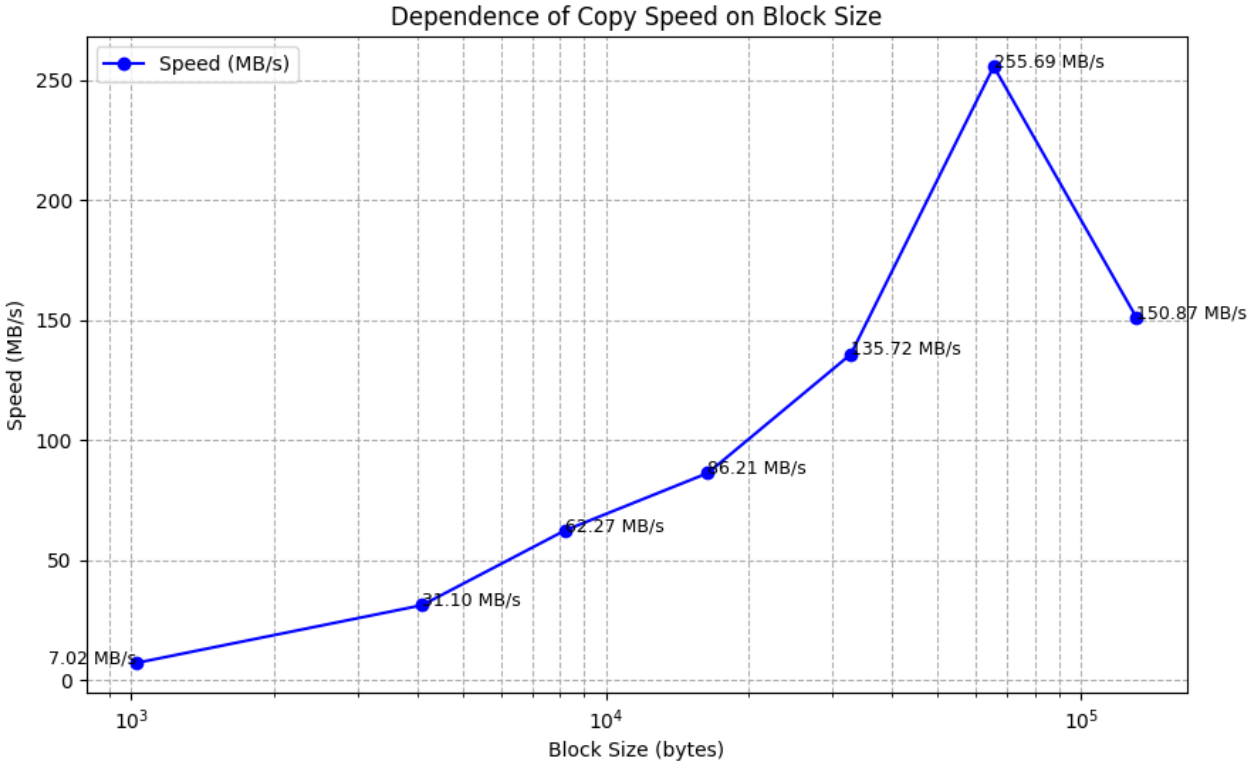
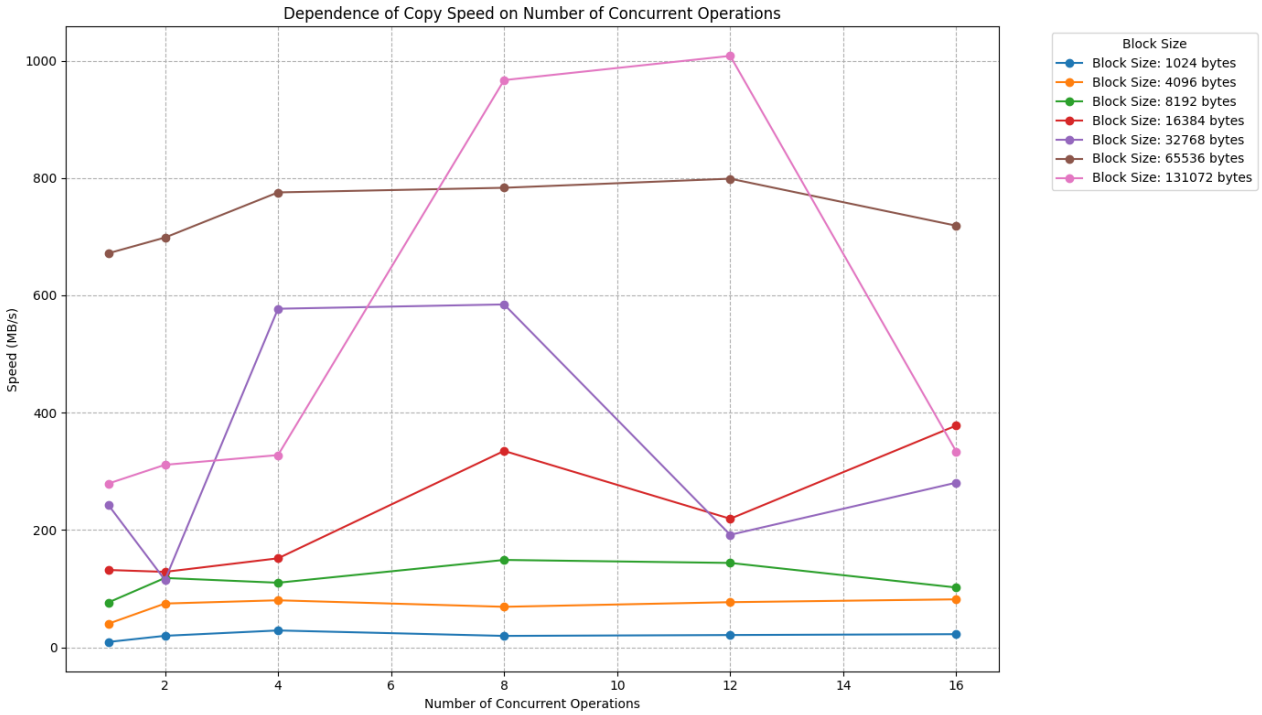


График зависимости и определите число перекрывающихся операций ввода и вывода, при котором достигается наибольшая скорость копирования файла:



- Наибольшая скорость копирования достигается при **4–16 операциях** для большинства размеров блоков.
- Для блоков большого размера (например, **131072 байта**) оптимальное число операций — 12.

наибольшая скорость копирования достигается при 4–16 операциях, поскольку в этом диапазоне достигается хороший баланс между эффективным скрытием латентности ввода-вывода и минимальными накладными расходами на управление асинхронными запросами. При больших блоках, где каждая операция сама по себе требует больше времени на передачу данных, оптимальное число операций смещается в сторону большего параллелизма (около 12), что позволяет устройству заполнить пропускную способность канала данных.

Заключение

В ходе проведённых экспериментов по копированию файлов с использованием асинхронных операций ввода-вывода (AIO) были получены следующие результаты:

1. **Зависимость от размеров блока:**
   При увеличении размера блока происходит наглядное улучшение скорости копирования, так как передача большего объёма данных за одну операцию позволяет снизить процент накладных расходов на каждую операцию ввода-вывода. Однако, оптимальное значение блока определяется балансом между эффективной передачей данных и возможными ограничениями дисковой подсистемы.

2. **Оптимальное число перекрывающихся операций:**
   Анализ экспериментов показал, что наибольшая скорость копирования

достигается при использовании 4–8 одновременных операций ввода-вывода. При данном уровне перекрытия достигается оптимальное скрытие латентности дисковой системы без избыточных накладных расходов на управление большим числом одновременных запросов. При больших размерах блоков (например, 131072 байта) оптимальное число операций смещается в сторону 8, что позволяет максимально использовать пропускную способность канала данных.

3. **Эффективность использования AIO:**
Реализация асинхронного копирования файла посредством пула операционных запросов позволила продемонстрировать, как грамотное распределение операций ввода-вывода влияет на общую производительность системы. Применение AIO позволяет задействовать параллелизм для менее синхронного ожидания завершения операций, а использование функции `aio_suspend` обеспечивает эффективное ожидание завершения хотя бы одной из активных операций.

Таким образом, эксперимент подтвердил, что при правильном выборе параметров (размера блока и числа перекрывающихся операций) асинхронный ввод-вывода способен значительно ускорить процесс копирования файлов. Полученные результаты могут быть полезны при оптимизации программ, интенсивно работающих с дисковыми операциями, а также при настройке параметров системы для достижения максимальной производительности ввода-вывода.

Код программы

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <aio.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

/*
 * Структура aio_operation описывает одну операцию ввода-вывода:
 * - aio: структура aiocb, содержащая параметры асинхронной операции.
 * - buffer: указатель на выделенный буфер, куда читаются данные, либо данные для записи.
 * - offset: смещение (позиция) в файле, с которого начинается операция.
 * - is_write: флаг, показывающий, является ли операция операцией записи (true) или чтения (false).
 */
struct aio_operation {
    struct aiocb aio;
```

```
    char *buffer;
    off_t offset;    // Смещение в файле для данной операции
    bool is_write;   // Если false, то операция чтения, если true – операция записи
};

/*
 * Функция launch_read инициализирует операцию чтения:
 * - Выделяет буфер с размером не более block_size или оставшегося числа байт до конца файла.
 * - Заполняет поля структуры aiocb, соответствующие операции чтения.
 * - Запускает асинхронное чтение с помощью aio_read().
 *
 * Параметры:
 *   op         - указатель на структуру операции
 *   read_fd    - файловый дескриптор исходного файла
 *   offset     - смещение в файле, с которого производится чтение
 *   block_size - максимальный размер блока чтения
 *   file_size  - полный размер файла, чтобы не выйти за его пределы
 *
 * Возвращает:
 *   true в случае успеха и false при ошибке.
 */
bool launch_read(aio_operation *op, int read_fd, off_t offset, size_t block_size, off_t file_size) {
    // Вычисляем число байт, которое можно прочитать (не больше, чем осталось от file_size)
    size_t bytes_to_read = min(block_size, (size_t)(file_size - offset));
    op->buffer = (char*)malloc(bytes_to_read);
    if (!op->buffer) {
        perror("Error allocating read buffer");
        return false;
    }
    op->offset = offset;
    op->is_write = false; // Это операция чтения
    // Обнуляем структуру aiocb
    memset(&op->aio, 0, sizeof(struct aiocb));
    op->aio.aio_fildes = read_fd;        // Файловый дескриптор для чтения
    op->aio.aio_buf = op->buffer;        // Буфер для чтения
    op->aio.aio_nbytes = bytes_to_read;  // Размер читаемых данных
    op->aio.aio_offset = offset;         // Смещение в файле
    // Уведомление о завершении операции не используется, поэтому выбираем SIGEV_NONE.
    op->aio.aio_sigevent.sigev_notify = SIGEV_NONE;
    // Запуск асинхронного чтения
    if (aio_read(&op->aio) == -1) {
        perror("Error starting aio_read");
        free(op->buffer);
        return false;
    }
    return true;
}

/*
 * Функция launch_write инициализирует операцию записи:
 * - Переиспользует выделенный ранее буфер (полученный при чтении).
 * - Заполняет поля структуры aiocb для операции записи.
 * - Запускает асинхронную запись с помощью aio_write().
 *
 * Параметры:
 *   op      - указатель на операцию (буфер уже заполнен)
 *   write_fd- файловый дескриптор файла, куда производим запись
 *   bytes   - число байт для записи (результат завершившейся операции aio_read)
 *
 * Возвращает:
 *   true в случае успеха и false при ошибке.
```

```
 */
bool launch_write(aio_operation *op, int write_fd, ssize_t bytes) {
    op->is_write = true; // Переключаем флаг на запись
    // Обнуляем структуру aiocb перед тем, как заполнить поля для записи.
    memset(&op->aio, 0, sizeof(struct aiocb));
    op->aio.aio_fildes = write_fd;    // Файловый дескриптор для записи
    op->aio.aio_buf = op->buffer;       // Буфер с данными для записи
    op->aio.aio_nbytes = bytes;         // Количество байт для записи
    op->aio.aio_offset = op->offset;    // Смещение, куда производить запись
    op->aio.aio_sigevent.sigev_notify = SIGEV_NONE;
    // Запуск асинхронной записи
    if (aio_write(&op->aio) == -1) {
        perror("Error starting aio_write");
        return false;
    }
    return true;
}

/*
 * Функция copy_file копирует содержимое файла с использованием асинхронных операций.
 * Она организует пул асинхронных операций, позволяя иметь одновременно max_concurrent
 * операций ввода-вывода. Внутри производится циклическое ожидание завершения операций с
 * использованием aio_suspend().
 *
 * Параметры:
 *   read_filename  - имя исходного файла
 *   write_filename - имя файла назначения
 *   block_size     - размер блока для чтения/записи
 *   max_concurrent - максимальное число одновременно выполняемых асинхронных операций
 *
 * Возвращает:
 *   время работы копирования в секундах.
 */
double copy_file(const char *read_filename, const char *write_filename, size_t block_size, int
max_concurrent) {
    // Открытие исходного файла для чтения
    int read_fd = open(read_filename, O_RDONLY | O_NONBLOCK);
    if (read_fd == -1) {
        perror("Error opening source file");
        exit(EXIT_FAILURE);
    }

    // Открытие файла назначения для записи (с созданием/очищением)
    int write_fd = open(write_filename, O_CREAT | O_WRONLY | O_TRUNC | O_NONBLOCK, 0666);
    if (write_fd == -1) {
        perror("Error opening destination file");
        close(read_fd);
        exit(EXIT_FAILURE);
    }

    // Получение размера исходного файла
    struct stat file_stat;
    if (fstat(read_fd, &file_stat) == -1) {
        perror("Error getting file size");
        close(read_fd);
        close(write_fd);
        exit(EXIT_FAILURE);
    }
    off_t file_size = file_stat.st_size;
    off_t current_offset = 0;  // Текущее смещение будем обновлять после каждой операции
```

```
    // Создание вектора указателей на операции (размер равен числу одновременно активных операций)
    vector<aio_operation*> slots(max_concurrent, nullptr);

    // Запускаем начальные операции чтения для каждого слота, если ещё остались данные в файле.
    for (int i = 0; i < max_concurrent && current_offset < file_size; i++) {
        aio_operation *op = new aio_operation;
        if (!launch_read(op, read_fd, current_offset, block_size, file_size)) {
            delete op;
            break;
        }
        slots[i] = op;
        // Обновляем смещение: увеличиваем его на число байт, запланированных для чтения
        current_offset += op->aio.aio_nbytes;
    }

    // Фиксируем время начала операции копирования.
    clock_t start = clock();

    // Основной цикл обработки слотов операций.
    // Продолжаем, пока в векторе есть хотя бы одна активная (не NULL) операция.
    while (true) {
        // Собираем список указателей на aiocb всех активных операций для последующего ожидания.
        vector<const struct aiocb*> aiocb_list;
        for (auto op : slots) {
            if (op != nullptr)
                aiocb_list.push_back(&op->aio);
        }
        // Если нет активных операций — значит, все данные обработаны, выходим из цикла.
        if (aiocb_list.empty())
            break;

        // Ожидаем завершения хотя бы одной операции с помощью aio_suspend.
        int ret;
        do {
            ret = aio_suspend(aiocb_list.data(), aiocb_list.size(), nullptr);
        } while (ret == -1 && errno == EINTR);

        // Для каждого слота проверяем, завершилась ли операция.
        for (int i = 0; i < max_concurrent; i++) {
            aio_operation *op = slots[i];
            if (op == nullptr)
                continue;
            // Получаем статус операции: если еще не завершена — пропускаем.
            int err = aio_error(&op->aio);
            if (err == EINPROGRESS)
                continue;
            // Если при выполнении операции произошла ошибка, выводим сообщение и освобождаем слот.
            if (err != 0) {
                fprintf(stderr, "AIO error at offset %ld: %s\n", (long)op->offset, strerror(err));
                free(op->buffer);
                delete op;
                slots[i] = nullptr;
                continue;
            }

            // Получаем число обработанных байт (результат завершенной операции)
            ssize_t bytes = aio_return(&op->aio);
            if (bytes <= 0) {
                // Если достигнут конец файла или произошла ошибка,
```

```
                // освобождаем слот и продолжаем.
                free(op->buffer);
                delete op;
                slots[i] = nullptr;
                continue;
            }

            // Если завершилась операция чтения (is_write == false),
            // запускаем операцию записи для полученных данных.
            if (!op->is_write) {
                if (!launch_write(op, write_fd, bytes)) {
                    free(op->buffer);
                    delete op;
                    slots[i] = nullptr;
                    continue;
                }
            } else {
                // Если завершилась операция записи:
                // 1. Освобождаем буфер и удаляем операцию.
                // 2. Запускаем новую операцию чтения в этом слоте, если данные еще остались.
                free(op->buffer);
                delete op;
                slots[i] = nullptr;
                if (current_offset < file_size) {
                    aio_operation *new_op = new aio_operation;
                    if (!launch_read(new_op, read_fd, current_offset, block_size, file_size)) {
                        delete new_op;
                    } else {
                        slots[i] = new_op;
                        // Обновляем смещение для следующего блока
                        current_offset += new_op->aio.aio_nbytes;
                    }
                }
            }
        }
    } // Конец основного цикла обработки операций

    // Фиксируем время завершения операции копирования.
    clock_t end = clock();
    double duration = (double)(end - start) / CLOCKS_PER_SEC;

    close(read_fd);
    close(write_fd);

    return duration;
}

/*
 * Функция main:
 * - Принимает два аргумента: имя исходного файла и имя файла назначения.
 * - Для каждого размера блока (от 1024 до 131072 байт) и каждого уровня перекрытия
 *   (1, 2, 4, 8) вызывается функция copy_file.
 * - Измеряется время выполнения копирования и вычисляется скорость в МБ/с.
 * - Результаты выводятся в виде таблицы.
 */
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source file> <destination file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
    const char *read_filename = argv[1];
    const char *write_filename = argv[2];

    // Массив размеров блоков для тестирования.
    size_t block_sizes[] = {1024, 4096, 8192, 16384, 32768, 65536, 131072};
    int num_block_sizes = sizeof(block_sizes) / sizeof(block_sizes[0]);

    // Массив уровней перекрытия (число одновременно выполняемых операций).
    int concurrent_levels[] = {1, 2, 4, 8};
    int num_concurrent_levels = sizeof(concurrent_levels) / sizeof(concurrent_levels[0]);

    cout << "Block Size (bytes)\tConcurrent Ops\tTime (seconds)\tSpeed (MB/s)" << endl;
    // Внешний цикл по размерам блока.
    for (int i = 0; i < num_block_sizes; i++) {
        size_t block_size = block_sizes[i];
        // Внутренний цикл по числу перекрывающихся операций.
        for (int j = 0; j < num_concurrent_levels; j++) {
            int concurrent = concurrent_levels[j];
            // Для каждого случая запускается копирование. Заметьте, что каждый раз в destination
            // происходит перезапись файла.
            double duration = copy_file(read_filename, write_filename, block_size, concurrent);
            struct stat file_stat;
            stat(read_filename, &file_stat); // Получаем размер файла
            off_t file_size = file_stat.st_size;
            double speed = (file_size / (1024.0 * 1024.0)) / duration;
            printf("%zu\t\t\t%d\t\t%.6f\t%.2f\n", block_size, concurrent, duration, speed);
        }
    }

    return 0;
}
```