Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

Студент: [Денисов Андрей]

Группа: М8О–203БВ–24

Вариант: 11

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2025.

**Постановка задачи**

## Цель работы

Целью является приобретение практических навыков в:

* Управление потоками в ОС
* Обеспечение синхронизации между потоками

## Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

В отчете привести исследование зависимости ускорения и эффективности алгоритма от входных данных и количества потоков. Получившиеся результаты необходимо объяснить.

**Общие сведения о программе**

Структуры данных:

TaskQueue - очередь задач для распределения работы между потоками. Содержит буфер для хранения номеров строк, мьютекс для синхронизации и условную переменную для ожидания задач.

Функции управления очередью:

tq\_init() - инициализация очереди задач, выделение памяти  
tq\_destroy() - освобождение ресурсов очереди  
tq\_push() - добавление задачи в очередь (потокобезопасная)  
tq\_pop() - извлечение задачи из очереди (блокирующая)  
tq\_shutdown() - сигнал завершения работы очереди

Вспомогательные функции:

cmp\_int() - функция сравнения для qsort  
print\_usage() - вывод справки по использованию программы

Основные функции:

worker\_main() - функция рабочего потока, обрабатывает строки матрицы медианным фильтром, обновляет статистику активности  
main() - главная функция: разбор аргументов, инициализация, создание потоков, организация итераций обработки, вывод результатов

Глобальные переменные:

g\_in, g\_out - указатели на входной и выходной буферы  
g\_R, g\_C - размеры матрицы  
g\_win - размер окна фильтра  
g\_active, g\_max\_active - атомарные счетчики активности потоков  
completed - счетчик завершенных задач  
completed\_mutex, completed\_cond - для синхронизации завершения

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Чтение входной матрицы

2. Создание пула потоков и очереди задач

3. Для каждой итерации K:

   - Распределение строк матрицы между потоками

   - Каждый поток обрабатывает свою строку:

     - Для каждого элемента применяет медианную фильтрацию в окрестности window\_size × window\_size

     - Использует padding по границам матрицы

   - Синхронизация потоков

4. Запись результата

**Основные файлы программы**

// median filter

#define \_POSIX\_C\_SOURCE 200809L

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>

#include <stdatomic.h>

// Очередь задач

typedef struct {

    int \*buf;           // Буфер для хранения задач (номеров строк)

    int capacity;       // Максимальная вместимость очереди

    int head, tail, size; // Указатели начала/конца и текущий размер очереди

    pthread\_mutex\_t mutex; // Мьютекс для синхронизации доступа

    pthread\_cond\_t cond\_nonempty; // Условная переменная для уведомления о новых задачах

    int shutdown;       // Флаг завершения работы очереди

} TaskQueue;

/\* Глобальные переменные \*/

static int \*in = NULL;    // Входной буфер матрицы

static int \*out = NULL;   // Выходной буфер матрицы

static int numberOfRows = 0;         // Количество строк матрицы

static int numberOfColumns = 0;         // Количество столбцов матрицы

static int sizeWindow = 3;       // Размер окна фильтра (по умолчанию 3x3)

// Атомарные переменные для отслеживания активности потоков

static atomic\_int currentNumberOfActive = 0;      // Текущее количество активных потоков

static atomic\_int maxActive = 0;  // Максимальное количество одновременно активных потоков

// Переменные для синхронизации завершения итераций

static int completed = 0;            // Счетчик завершенных строк

static pthread\_mutex\_t completed\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

static pthread\_cond\_t completed\_cond = PTHREAD\_COND\_INITIALIZER;

// Инициализация очереди задач

static int tq\_init(TaskQueue \*q, int capacity) {

    q->buf = malloc(sizeof(int) \* capacity); // Выделение памяти под буфер

    if (!q->buf) return -1;                  // Проверка на ошибку

    q->capacity = capacity;

    q->head = q->tail = q->size = 0;

    q->shutdown = 0;

    pthread\_mutex\_init(&q->mutex, NULL);     // Инициализация мьютекса

    pthread\_cond\_init(&q->cond\_nonempty, NULL); // Инициализация условной переменной

    return 0;

}

// Освобождение ресурсов очереди

static void tq\_destroy(TaskQueue \*q) {

    if (!q) return;

    free(q->buf);                            // Освобождение памяти

    pthread\_mutex\_destroy(&q->mutex);        // Деструкция мьютекса

    pthread\_cond\_destroy(&q->cond\_nonempty); // Деструкция условной переменной

}

// Добавление в очередь задач

static void tq\_push(TaskQueue \*q, int row) {

    pthread\_mutex\_lock(&q->mutex);           // Блокировка мьютекса

    q->buf[q->tail] = row;                   // Добавление номера строки

    q->tail = (q->tail + 1) % q->capacity;   // Циклический сдвиг указателя

    q->size++;                               // Увеличение размера очереди

    pthread\_cond\_signal(&q->cond\_nonempty);  // Сигнал о наличии задачи

    pthread\_mutex\_unlock(&q->mutex);         // Разблокировка мьютекса

}

// Извлечение задачи из очереди

static int tq\_pop(TaskQueue \*q, int \*out\_row) {

    pthread\_mutex\_lock(&q->mutex);           // Блокировка мьютекса

    while (q->size == 0 && !q->shutdown) {   // Ожидание, пока очередь не станет непустой

        pthread\_cond\_wait(&q->cond\_nonempty, &q->mutex);

    }

    if (q->size == 0 && q->shutdown) {       // Проверка на завершение

        pthread\_mutex\_unlock(&q->mutex);

        return 0; // Нет задач, завершение

    }

    \*out\_row = q->buf[q->head];              // Извлечение номера строки

    q->head = (q->head + 1) % q->capacity;   // Циклический сдвиг указателя

    q->size--;                               // Уменьшение размера очереди

    pthread\_mutex\_unlock(&q->mutex);         // Разблокировка мьютекса

    return 1;

}

// Завершение работы очереди

static void tq\_shutdown(TaskQueue \*q) {

    pthread\_mutex\_lock(&q->mutex);

    q->shutdown = 1;                           // Установка флага завершения

    pthread\_cond\_broadcast(&q->cond\_nonempty); // Сигнал всем потокам

    pthread\_mutex\_unlock(&q->mutex);

}

// Компаратор для сортировки значений

static int cmp\_int(const void \*a, const void \*b) {

    int ia = \*(const int\*)a;

    int ib = \*(const int\*)b;

    return (ia > ib) - (ia < ib); // Сравнение для qsort

}

// Структура аргументов для потока

typedef struct {

    TaskQueue \*queue; // Указатель на очередь задач

} WorkerArg;

// Основная функция потока

static void \*worker\_main(void \*vp) {

    WorkerArg \*warg = (WorkerArg\*)vp;

    TaskQueue \*q = warg->queue;

    // Выделение памяти под временное хранение значений окна

    int wsize = sizeWindow \* sizeWindow;

    int \*vals = malloc(sizeof(int) \* wsize);

    if (!vals) {

        fprintf(stderr, "Worker: malloc failed\n");

        return NULL;

    }

    while (1) {

        int row;

        if (!tq\_pop(q, &row)) break; // Завершение, если нет задач

        // Отслеживание активных потоков

        int cur = atomic\_fetch\_add(&currentNumberOfActive, 1) + 1;

        int prev;

        do {

            prev = atomic\_load(&maxActive);

            if (cur <= prev) break;

        } while (!atomic\_compare\_exchange\_weak(&maxActive, &prev, cur));

        // Обработка строки `row`

        int half = sizeWindow / 2;

        for (int col = 0; col < numberOfColumns; ++col) {

            int idx = 0;

            for (int dr = -half; dr <= half; ++dr) {

                int rr = row + dr;

                if (rr < 0) rr = 0;

                if (rr >= numberOfRows) rr = numberOfRows - 1;

                for (int dc = -half; dc <= half; ++dc) {

                    int cc = col + dc;

                    if (cc < 0) cc = 0;

                    if (cc >= numberOfColumns) cc = numberOfColumns - 1;

                    vals[idx++] = in[rr \* numberOfColumns + cc]; // Сбор значений окна

                }

            }

            qsort(vals, (size\_t)idx, sizeof(int), cmp\_int); // Сортировка

            int median = vals[idx/2]; // Нахождение медианы

            out[row \* numberOfColumns + col] = median; // Запись результата

        }

        // Уменьшение счетчика активных потоков

        atomic\_fetch\_sub(&currentNumberOfActive, 1);

        // Уведомление основного потока о завершении строки

        pthread\_mutex\_lock(&completed\_mutex);

        completed++;

        if (completed == numberOfRows) {

            pthread\_cond\_signal(&completed\_cond);

        }

        pthread\_mutex\_unlock(&completed\_mutex);

    }

    free(vals); // Освобождение памяти

    return NULL;

}

// Вывод помощи при использовании

static void print\_usage(const char \*prog) {

    fprintf(stderr,

        "Usage: %s [-m max\_threads] -k K -w window\_size [-i infile] [-o outfile]\n"

        "  -m max\_threads   максимальное количество потоков (по умолчанию: количество ядер)\n"

        "  -k K             количество применений фильтра (>=1)\n"

        "  -w window\_size   размер окна (нечетное число, например 3,5,7)\n"

        "  -i infile        входной файл (по умолчанию: stdin)\n"

        "  -o outfile       выходной файл (по умолчанию: stdout)\n",

        prog);

}

// Основная функция программы

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

    int opt;

    int max\_threads = 0; // Максимальное количество потоков

    int K = -1;          // Количество итераций фильтра

    int win = -1;        // Размер окна

    char \*infile = NULL, \*outfile = NULL;

    // Парсинг аргументов командной строки

    while ((opt = getopt(argc, argv, "m:k:w:i:o:")) != -1) {

        switch (opt) {

            case 'm': max\_threads = atoi(optarg); break;

            case 'k': K = atoi(optarg); break;

            case 'w': win = atoi(optarg); break;

            case 'i': infile = optarg; break;

            case 'o': outfile = optarg; break;

            default:

                print\_usage(argv[0]);

                return 1;

        }

    }

    // Проверка корректности аргументов

    if (K < 1 || win < 1 || (win % 2 == 0)) {

        fprintf(stderr, "Ошибка: K должно быть >=1, window\_size должно быть нечетным.\n");

        print\_usage(argv[0]);

        return 1;

    }

    sizeWindow = win;

    // Определение количества потоков по умолчанию

    if (max\_threads <= 0) {

        long procs = sysconf(\_SC\_NPROCESSORS\_ONLN);

        max\_threads = (procs > 0) ? (int)procs : 4;

    }

    if (max\_threads < 1) max\_threads = 1;

    // Открытие входного и выходного файлов

    FILE \*fin = stdin;

    FILE \*fout = stdout;

    if (infile) {

        fin = fopen(infile, "r");

        if (!fin) { perror("fopen infile"); return 1; }

    }

    if (outfile) {

        fout = fopen(outfile, "w");

        if (!fout) { perror("fopen outfile"); if (infile) fclose(fin); return 1; }

    }

    // Чтение размеров матрицы

    if (fscanf(fin, "%d %d", &numberOfRows, &numberOfColumns) != 2) {

        fprintf(stderr, "Не удалось прочитать R C из входного файла\n");

        if (infile) fclose(fin);

        if (outfile) fclose(fout);

        return 1;

    }

    if (numberOfRows <= 0 || numberOfColumns <= 0) {

        fprintf(stderr, "Неверный размер матрицы\n");

        if (infile) fclose(fin);

        if (outfile) fclose(fout);

        return 1;

    }

    // Выделение памяти под буферы

    size\_t total = (size\_t)numberOfRows \* (size\_t)numberOfColumns;

    int \*buf1 = malloc(sizeof(int) \* total);

    int \*buf2 = malloc(sizeof(int) \* total);

    if (!buf1 || !buf2) {

        fprintf(stderr, "Ошибка выделения памяти\n");

        free(buf1); free(buf2);

        if (infile) fclose(fin);

        if (outfile) fclose(fout);

        return 1;

    }

    // Чтение матрицы из файла

    for (size\_t i = 0; i < total; ++i) {

        if (fscanf(fin, "%d", &buf1[i]) != 1) {

            fprintf(stderr, "Недостаточно элементов матрицы в входном файле\n");

            free(buf1); free(buf2);

            if (infile) fclose(fin);

            if (outfile) fclose(fout);

            return 1;

        }

    }

    if (infile) fclose(fin);

    // Инициализация очереди задач

    TaskQueue queue;

    if (tq\_init(&queue, numberOfRows) != 0) {

        fprintf(stderr, "Ошибка инициализации очереди задач\n");

        free(buf1); free(buf2);

        return 1;

    }

    // Создание массивов для потоков

    pthread\_t \*workers = malloc(sizeof(pthread\_t) \* max\_threads);

    WorkerArg \*wargs = malloc(sizeof(WorkerArg) \* max\_threads);

    if (!workers || !wargs) {

        fprintf(stderr, "Ошибка выделения памяти для потоков\n");

        free(buf1); free(buf2); free(workers); free(wargs);

        tq\_destroy(&queue);

        return 1;

    }

    // Создание потоков

    for (int i = 0; i < max\_threads; ++i) {

        wargs[i].queue = &queue;

        int rc = pthread\_create(&workers[i], NULL, worker\_main, &wargs[i]);

        if (rc != 0) {

            fprintf(stderr, "Ошибка создания потока %d: %s\n", i, strerror(rc));

            workers[i] = 0;

        }

    }

    // Выполнение K итераций фильтрации

    int \*cur = buf1;

    int \*next = buf2;

    for (int iter = 0; iter < K; ++iter) {

        // Установка указателей на текущие буферы

        in = cur;

        out = next;

        // Сброс счетчика завершенных строк

        pthread\_mutex\_lock(&completed\_mutex);

        completed = 0;

        pthread\_mutex\_unlock(&completed\_mutex);

        // Добавление всех строк в очередь

        for (int r = 0; r < numberOfRows; ++r) tq\_push(&queue, r);

        // Ожидание завершения всех строк

        pthread\_mutex\_lock(&completed\_mutex);

        while (completed < numberOfRows) {

            pthread\_cond\_wait(&completed\_cond, &completed\_mutex);

        }

        pthread\_mutex\_unlock(&completed\_mutex);

        // Переключение буферов

        int \*tmp = cur; cur = next; next = tmp;

    }

    // Завершение работы потоков

    tq\_shutdown(&queue);

    for (int i = 0; i < max\_threads; ++i) {

        if (workers[i]) pthread\_join(workers[i], NULL);

    }

    // Вывод информации о параллелизме

    fprintf(stderr, "Настроено потоков: %d\n", max\_threads);

    fprintf(stderr, "Максимальное одновременное количество активных потоков: %d\n", atomic\_load(&maxActive));

    // Запись результата в файл

    fprintf(fout, "%d %d\n", numberOfRows, numberOfColumns);

    for (int i = 0; i < numberOfRows; ++i) {

        for (int j = 0; j < numberOfColumns; ++j) {

            fprintf(fout, "%d%c", cur[i\*numberOfColumns + j], (j+1==numberOfColumns) ? '\n' : ' ');

        }

    }

    if (outfile) fclose(fout);

    // Освобождение ресурсов

    free(buf1); free(buf2);

    free(workers); free(wargs);

    tq\_destroy(&queue);

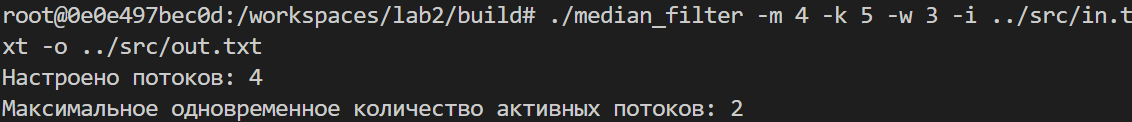
    pthread\_mutex\_destroy(&completed\_mutex);

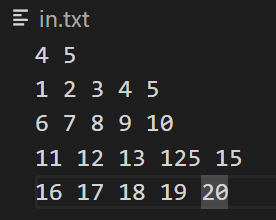
    pthread\_cond\_destroy(&completed\_cond);

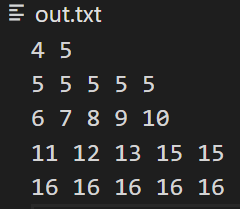
    return 0;

}

**Пример работы**



****

****

**Вывод**

В ходе выполнения лабораторной работы был разработан многопоточный медианный фильтр для обработки матриц. Реализация использует пул потоков и очередь задач для параллельной обработки строк матрицы.

Программа демонстрирует эффективное использование механизмов синхронизации: мьютексов для защиты разделяемых данных и условных переменных для координации работы потоков. Очередь задач обеспечивает равномерное распределение нагрузки между рабочими потоками.

Атомарные переменные используются для отслеживания активности потоков, что позволяет анализировать степень параллелизма. Программа корректно обрабатывает граничные случаи и поддерживает многократное применение фильтра.

Результаты показывают, что многопоточный подход позволяет эффективно ускорить обработку данных за счет параллельного выполнения операций над различными строками матрицы.