Московский авиационный институт

(национальный исследовательский университет)

# Факультет информационных технологий и прикладной математики

**Кафедра вычислительной математики и программирования**

## Лабораторная работа № 2 по курсу «Операционные системы»

**Управление потоками в ОС.**

**Обеспечение синхронизации между потоками**

Студент: Кунавин К. В. Преподаватель: Миронов Е. С.

Группа: М8О-203Б-23

Дата: Оценка: Подпись:

**Москва, 2024**

# Условие

Составить программу на языке С/C++, 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.

# Задание

Рассчитать детерминант матрицы (используя определение детерминанта).

# Метод решения

Реализована рекурсивная функция на C++ для вычисления определителя n×n-матрицы по формуле разложения по первой строке. Для ускорения использовалась многопоточность: при разбиении на миноры программа запускала потоки (через pthread), чтобы каждая часть вычислялась параллельно. Число одновременно работающих потоков ограничивалось специальным семафором или пулом заданий, чтобы избежать слишком большого распараллеливания и перегрузки системы.

# Код программы

TDeterminantCalculator.h

#ifndef TDETERMINANT\_CALCULATOR\_H

#define TDETERMINANT\_CALCULATOR\_H

#include "TMatrix.h"

#include <queue>

#include <pthread.h>

#include <memory>

class TDeterminantCalculator {

public:

    TDeterminantCalculator(const TMatrix& Matrix, int numThreads);

    ~TDeterminantCalculator();

    // Метод для вычисления определителя

    double Compute();

private:

    // Задача для потока

    struct TTask {

        TMatrix Matrix;

        double element;

        TTask(const TMatrix& Mat, double e);

    };

    // Статическая функция-обёртка для потоков

    static void\* WorkerHelper(void\* arg);

    // Основная функция потока

    void\* Worker();

    // Методы для QueueTask

    void PushTask(std::unique\_ptr<TTask> Task);

    bool PopTask(std::unique\_ptr<TTask>& Task);

    void Shutdown();

    TMatrix GetMinor(const TMatrix& Matrix, int row, int col);

    const TMatrix& Matrix\_;

    int numThreads\_;

    double globalDeterminant\_;

    int activeTasks\_;

    bool stop\_;

    std::queue<std::unique\_ptr<TTask>> taskQueue\_;

    // Синхронизация

    pthread\_mutex\_t queueMutex\_;

    pthread\_cond\_t queueCond\_;

    pthread\_mutex\_t determinantMutex\_;

    pthread\_mutex\_t activeTasksMutex\_;

    pthread\_cond\_t activeTasksCond\_;

};

#endif

TMatrix.h

#ifndef INPUT\_TMATRIX\_H

#define INPUT\_TMATRIX\_H

#include <vector>

using TMatrix = std::vector<std::vector<double>>;

TMatrix InputTMatrix();

#endif

TDeterminantCalculator.cpp

#include "TDeterminantCalculator.h"

#include <iostream>

#include <cstdlib>

#include <cstring>

TDeterminantCalculator::TTask::TTask(const TMatrix& Mat, double e)

    : Matrix(Mat), element(e) {}

TDeterminantCalculator::TDeterminantCalculator(const TMatrix& Matrix, int numThreads)

    : Matrix\_(Matrix), numThreads\_(numThreads), globalDeterminant\_(0.0), activeTasks\_(0), stop\_(false) {

    pthread\_mutex\_init(&queueMutex\_, nullptr);

    pthread\_cond\_init(&queueCond\_, nullptr);

    pthread\_mutex\_init(&determinantMutex\_, nullptr);

    pthread\_mutex\_init(&activeTasksMutex\_, nullptr);

    pthread\_cond\_init(&activeTasksCond\_, nullptr);

}

TDeterminantCalculator::~TDeterminantCalculator() {

    pthread\_mutex\_destroy(&queueMutex\_);

    pthread\_cond\_destroy(&queueCond\_);

    pthread\_mutex\_destroy(&determinantMutex\_);

    pthread\_mutex\_destroy(&activeTasksMutex\_);

    pthread\_cond\_destroy(&activeTasksCond\_);

}

void TDeterminantCalculator::PushTask(std::unique\_ptr<TTask> Task) {

    pthread\_mutex\_lock(&queueMutex\_);

    taskQueue\_.push(std::move(Task));

    pthread\_cond\_signal(&queueCond\_);

    pthread\_mutex\_unlock(&queueMutex\_);

}

bool TDeterminantCalculator::PopTask(std::unique\_ptr<TTask>& Task) {

    pthread\_mutex\_lock(&queueMutex\_);

    while (taskQueue\_.empty() && !stop\_) {

        pthread\_cond\_wait(&queueCond\_, &queueMutex\_);

    }

    if (stop\_ && taskQueue\_.empty()) {

        pthread\_mutex\_unlock(&queueMutex\_);

        return false;

    }

    Task = std::move(taskQueue\_.front());

    taskQueue\_.pop();

    pthread\_mutex\_unlock(&queueMutex\_);

    return true;

}

void TDeterminantCalculator::Shutdown() {

    pthread\_mutex\_lock(&queueMutex\_);

    stop\_ = true;

    pthread\_cond\_broadcast(&queueCond\_);

    pthread\_mutex\_unlock(&queueMutex\_);

}

TMatrix TDeterminantCalculator::GetMinor(const TMatrix& Matrix, int row, int col) {

    TMatrix Minor;

    int n = Matrix.size();

    for (int i = 0; i < n; ++i) {

        if (i == row) {

            continue;

        }

        std::vector<double> minorRow;

        for (int j = 0; j < n; ++j) {

            if (j == col) {

                continue;

            }

            minorRow.push\_back(Matrix[i][j]);

        }

        Minor.push\_back(minorRow);

    }

    return Minor;

}

// Статическая функция-обёртка для потоков

void\* TDeterminantCalculator::WorkerHelper(void\* arg) {

    return ((TDeterminantCalculator\*)arg)->Worker();

}

// Основная функция потока

void\* TDeterminantCalculator::Worker() {

    while (true) {

        std::unique\_ptr<TTask> Task;

        if (!PopTask(Task)) {

            break; // Очередь остановлена и задач больше нет

        }

        int n = Task->Matrix.size();

        if (n == 1) {

            // Базовый случай 1x1 матрица

            double det = Task->element \* Task->Matrix[0][0];

            pthread\_mutex\_lock(&determinantMutex\_);

            globalDeterminant\_ += det;

            pthread\_mutex\_unlock(&determinantMutex\_);

        }

        else if (n == 3) {

            // Базовый случайM 3x3 матрица

            double a = Task->Matrix[0][0];

            double b = Task->Matrix[0][1];

            double c = Task->Matrix[0][2];

            double d = Task->Matrix[1][0];

            double e = Task->Matrix[1][1];

            double f = Task->Matrix[1][2];

            double g = Task->Matrix[2][0];

            double h = Task->Matrix[2][1];

            double i = Task->Matrix[2][2];

            // По методу звездочки

            double det = Task->element \* (a\*(e\*i - f\*h) - b\*(d\*i - f\*g) + c\*(d\*h - e\*g));

            pthread\_mutex\_lock(&determinantMutex\_);

            globalDeterminant\_ += det;

            pthread\_mutex\_unlock(&determinantMutex\_);

        } else {

            // Создаем подзадачи

            int n\_subtasks = n;

            pthread\_mutex\_lock(&activeTasksMutex\_);

            activeTasks\_ += n\_subtasks;

            pthread\_mutex\_unlock(&activeTasksMutex\_);

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

                TMatrix Minor = GetMinor(Task->Matrix, 0, j); // Раскладываем по первой строке

                double element = Task->Matrix[0][j];

                int sign = ((j % 2) == 0) ? 1 : -1;

                // Вычисляем новое накопленное значение элемента

                double newElement = Task->element \* element \* sign;

                // Создаем новую задачу и добавляем ее в очередь

                std::unique\_ptr<TTask> newTask = std::make\_unique<TTask>(Minor, newElement);

                PushTask(std::move(newTask));

            }

        }

        pthread\_mutex\_lock(&activeTasksMutex\_);

        activeTasks\_--;

        int remaining\_tasks = activeTasks\_;

        if (remaining\_tasks == 0) {

            // Уведомляем главный поток о завершении всех задач

            pthread\_cond\_signal(&activeTasksCond\_);

        }

        pthread\_mutex\_unlock(&activeTasksMutex\_);

    }

    return nullptr;

}

double TDeterminantCalculator::Compute() {

    globalDeterminant\_ = 0.0;

    activeTasks\_ = 0;

    stop\_ = false;

    // Создание потоков

    std::vector<pthread\_t> threads(numThreads\_);

    for (int i = 0; i < numThreads\_; ++i) {

        int ret = pthread\_create(&threads[i], nullptr, &TDeterminantCalculator::WorkerHelper, this);

        if (ret != 0) {

            std::cerr << "Ошибка создания потока: " << strerror(ret) << std::endl;

            exit(1);

        }

    }

    pthread\_mutex\_lock(&activeTasksMutex\_);

    activeTasks\_ = 1; // Корневая задача

    pthread\_mutex\_unlock(&activeTasksMutex\_);

    std::unique\_ptr<TTask> InitialTask = std::make\_unique<TTask>(Matrix\_, 1.0);

    PushTask(std::move(InitialTask));

    // Ожидаем завершения всех задач

    pthread\_mutex\_lock(&activeTasksMutex\_);

    while (activeTasks\_ > 0) {

        pthread\_cond\_wait(&activeTasksCond\_, &activeTasksMutex\_);

    }

    pthread\_mutex\_unlock(&activeTasksMutex\_);

    Shutdown(); // Останавливаем очередь задач

    // Ожидаем завершения всех потоков

    for (int i = 0; i < numThreads\_; ++i) {

        pthread\_join(threads[i], nullptr);

    }

    return globalDeterminant\_;

}

TMatrix.cpp

#include <iostream>

#include "TMatrix.h"

TMatrix InputTMatrix() {

    int n;

    std::cout << "Введите размер матрицы (n x n): ";

    std::cin >> n;

    if (n <= 0) {

        std::cerr << "Размер матрицы должен быть положительным." << std::endl;

        exit(1);

    }

    TMatrix Matrix(n, std::vector<double>(n));

    std::cout << "Введите элементы матрицы построчно:\n";

    for (int i = 0; i < n; ++i) {

        std::cout << "Строка " << i + 1 << ": ";

        for (int j = 0; j < n; ++j) {

            std::cin >> Matrix[i][j];

        }

    }

    return Matrix;

}

CMakeLists.txt

add\_executable(lab2 main.cpp src/TDeterminantCalculator.cpp include/TDeterminantCalculator.h src/TMatrix.cpp include/TMatrix.h)

target\_include\_directories(lab2 PRIVATE include)

main.cpp

#include <iostream>

#include "TDeterminantCalculator.h"

#include "TMatrix.h"

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

    // Обработка аргументов командной строки

    if (argc != 3 || std::string(argv[1]) != "-t") {

        std::cerr << "Использование: " << argv[0] << " -t <число\_потоков>" << std::endl;

        return 1;

    }

    int numThreads;

    try {

        numThreads = std::stoi(argv[2]);

    } catch (const std::exception& e) {

        std::cerr << "Ошибка: неверное значение числа потоков." << std::endl;

        return 1;

    }

    if (numThreads <= 0) {

        std::cerr << "Ошибка: число потоков должно быть положительным." << std::endl;

        return 1;

    }

    // Ввод матрицы пользователем

    TMatrix Matrix = InputTMatrix();

    // Вычисляем определитель (разложение по первой строке)

    TDeterminantCalculator Calculator(Matrix, numThreads);

    double determinant = Calculator.Compute();

    std::cout << "Определитель: " << determinant << std::endl;

    return 0;

}

lab2\_test.cpp

#include <TDeterminantCalculator.h>

#include <TMatrix.h>

#include <TDeterminantCalculator.h>

#include <TMatrix.h>

#include <gtest/gtest.h>

#include <chrono>

// Проверка корректности однопоточного варианта

TEST(DeterminantCalculatorTest, SingleThreadCorrectness) {

    // 1x1 матрица

    TMatrix Mat1 = {{5}};

    TDeterminantCalculator Calc1(Mat1, 1);

    double det1 = Calc1.Compute();

    EXPECT\_DOUBLE\_EQ(det1, 5.0);

    // 2x2 матрица

    TMatrix Mat2 = {

        {1, 2},

        {3, 4}

    };

    TDeterminantCalculator Calc2(Mat2, 1);

    double det2 = Calc2.Compute();

    EXPECT\_DOUBLE\_EQ(det2, -2.0);

    // 3x3 матрица

    TMatrix Mat3 = {

        {6, 1, 1},

        {4, -2, 5},

        {2, 8, 7}

    };

    TDeterminantCalculator Calc3(Mat3, 1);

    double det3 = Calc3.Compute();

    EXPECT\_DOUBLE\_EQ(det3, -306.0);

    // 4x4 матрица

    TMatrix Mat4 = {

        {3, 2, 0, 1},

        {4, 0, 1, 2},

        {3, 0, 2, 1},

        {9, 2, 3, 1}

    };

    TDeterminantCalculator Calc4(Mat4, 1);

    double det4 = Calc4.Compute();

    EXPECT\_DOUBLE\_EQ(det4, 24.0);

}

// Проверка соответствия результатов однопоточного и многопоточного вариантов

TEST(DeterminantCalculatorTest, SingleVsMultiThreadConsistency) {

    // 3x3 матрица

    TMatrix Mat = {

        {6, 1, 1},

        {4, -2, 5},

        {2, 8, 7}

    };

    // Однопоточный вариант

    TDeterminantCalculator CalcSingle(Mat, 1);

    double detSingle = CalcSingle.Compute();

    // Многопоточный вариант

    for (int i = 2; i <= 8; i += 2) {

        TDeterminantCalculator CalcMulti(Mat, i);

        double detMulti = CalcMulti.Compute();

        EXPECT\_DOUBLE\_EQ(detSingle, detMulti);

    }

}

// Проверка производительности многопоточного варианта

TEST(DeterminantCalculatorTest, MultiThreadPerformance) {

    // 11x11 матрица

    TMatrix Mat(11, std::vector<double>(11, 2.0));

    // Измеряем время для однопоточного варианта

    TDeterminantCalculator CalcSingle(Mat, 1);

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

    double detSingle = CalcSingle.Compute();

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

    std::chrono::duration<double> durationSingle = endSingle - startSingle;

    // Измеряем время для многопоточного варианта (например, 8 потоков)

    TDeterminantCalculator CalcMulti(Mat, 8);

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

    double detMulti = CalcMulti.Compute();

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

    std::chrono::duration<double> durationMulti = endMulti - startMulti;

    // Проверяем, что результаты совпадают

    EXPECT\_DOUBLE\_EQ(detSingle, detMulti);

    // Проверяем, что многопоточный вариант быстрее

    std::cout << "Однопоточный вариант: " << durationSingle.count() << " секунд\n";

    std::cout << "Многопоточный вариант: " << durationMulti.count() << " секунд\n";

    EXPECT\_LT(durationMulti.count(), durationSingle.count());

}

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

    testing::InitGoogleTest(&argc, argv);

    return RUN\_ALL\_TESTS();

}

# Выводы

Реализация многопоточного вычисления детерминанта позволила ускорить расчёты для больших матриц, но при слишком глубокой рекурсии и неразумном количестве потоков возникали накладные расходы. Контроль числа активных потоков и грамотная синхронизация (семафоры, мьютексы) стали ключевыми факторами стабильной и эффективной работы. Лабораторная работа показала, как многопоточность может существенно ускорить вычислительные задачи, но только при правильном управлении ресурсами.