Лабораторная работа №2

Предмет: Операционные системы
Преподаватель: Данилов А.С.
Студент: Сайранов Э.Р. ИТ-3,4-2024
Язык программирования: C++
Платформа: macOS/Linux

Дата: 18.11.2025

1. Постановка задачи **3**

1.1 Цель работы 3

1.2 Требования 3

2. Процесс решения **4**

2.1 Архитектура решения 4

2.1.1 Версия с потоками (matrix_threads.cpp) *4*

2.1.2 Версия с процессами (matrix_processes.cpp) *4*

2.2 Технические детали реализации 5

2.2.1 Структура ThreadData (потоки) *5*

2.2.2 Алгоритм умножения матриц *5*

2.2.3 Создание потоков *6*

2.2.4 Создание процессов и IPC *6*

2.2.5 Ожидание завершения *7*

2.3 Компиляция *7*

2.4 Тестирование *7*

3. Полученные графики **8**

4. Объяснение полученных результатов **9**

4.1 Поведение потоков 9

4.1.1 Малое количество потоков (1-4) *9*

4.1.2 Среднее количество потоков (4-8) *9*

4.1.3 Большое количество потоков (16+) *9*

4.2 Поведение процессов 10

4.2.1 Общие черты с потоками *10*

4.2.2 Дополнительные накладные расходы процессов *10*

4.3 Потоки vs процессы 11

4.3.1 Потоки быстрее на 10-50% *11*

4.3.2 Процессы более «чистые» с точки зрения изоляции *11*

5. Выводы **12**

5.1 Основные результаты 12

6. Приложения **13**

1. Постановка задачи

1.1 Цель работы

Реализовать программы для умножения матриц размером N×N с использованием:

- Многопоточности (POSIX threads)
- Многопроцессности (fork + IPC)

Провести анализ производительности и сравнить эффективность обоих подходов.

1.2 Требования

1. Функциональность:

- Корректное перемножение двух матриц N×N
- Поддержка произвольного количества потоков/процессов
- Передача параметров через аргументы командной строки
- Корректная работа на Linux/macOS

2. Производительность:

- Минимум 3 повтора для каждого теста
- Усреднение результатов
- Матрицы размером достаточным для получения результатов >2 сек
- Исключение времени генерации матриц из замеров

3. Анализ:

- Сравнение времени работы в зависимости от количества потоков/процессов
- Объяснение поведения производительности
- Построение сравнительных графиков
- Анализ ускорения (speedup)

4. Технические ограничения:

- Процессы создавать через fork()
- Использовать стандартные структуры данных (vector для матриц)
- Без специализированных библиотек (BLAS, Eigen и т.д.)
- На Python не использовать потоки (GIL ограничивает)

2. Процесс решения

2.1 Архитектура решения

2.1.1 Версия с потоками (matrix_threads.cpp)

Основной принцип: Разделение данных между потоками, общая память

Главный поток генерирует матрицы A и B, создает N рабочих потоков, ждет завершения всех потоков, выводит результаты.

Механизм синхронизации:

- std::thread - автоматически запускает поток
- thread.join() - главный поток ждем завершения рабочего потока
- Общая память между потоками

Распределение работы:

- Каждый поток получает диапазон строк для обработки
- Потоки могут писать в разные строки результирующей матрицы
- Нет конфликтов доступа (разные потоки → разные строки)

2.1.2 Версия с процессами (matrix_processes.cpp)

Основной принцип: изоляция данных, IPC для связи

Главный процесс генерирует матрицы A и B, создает pipes для каждого подпроцесса, fork() создает дочерних процессы, читает результаты из pipes, wait() ждет завершения подпроцессов

Механизм синхронизации:

- fork() — создает копию процесса
- pipe() — создает канал для обмена данными
- write() — дочерний процесс отправляет результаты в pipe
- read() — главный процесс читает результаты из pipe
- wait() — главный процесс ждет завершения дочерних процессов

Распределение работы:

- Каждый дочерний процесс вычисляет свои строки
- Результаты отправляются через pipe

- Главный процесс собирает результаты
- Каждый процесс завершается через exit(0)

2.2 Технические детали реализации

2.2.1 Структура ThreadData (потоки)

```
struct ThreadData {
    const Matrix* matrixA; — ссылка на матрицу A
    const Matrix* matrixB; — ссылка на матрицу B
    Matrix* resultMatrix; — указатель на результирующую матрицу
    int startRow; — начальная строка для этого потока
    int endRow; — конечная строка для этого потока
    int matrixSize; — размер матрицы N
}
```

Необходимость:

- Каждому потоку нужно знать, какие строки обрабатывать
- Ссылки на матрицы экономят память (не копируем всю матрицу)
- Указатель на результат позволяет писать в одну общую матрицу

2.2.2 Алгоритм умножения матриц

Для каждого элемента result[i][j]:
result[i][j] = sum(A[i][k] * B[k][j]) для всех k от 0 до N
Сложность: $O(N^3)$ для полного умножения
Сложность одной строки: $O(N^2)$

O(N^3):

- N строк в результирующей матрице
- N столбцов в результирующей матрице
- N операций умножения для каждого элемента
- Всего: N * N * N = N^3

Параллелизм:

- Обработка разных строк может идти параллельно
- Вычисления одной строки не зависят от других строк
- Идеально подходит для параллелизма на уровне данных

2.2.3 Создание потоков

```
std::vector<std::thread> threads;
for (int t = 0; t < numThreads; t++) {
    int startRow = t * rowsPerThread;
    int endRow = std::min((t + 1) * rowsPerThread, matrixSize);
    ThreadData* data = new ThreadData(&matrixA, &matrixB, &result, startRow, endRow,
matrixSize);
    threads.push_back(std::thread(multiplyMatrixRows, data));
}
```

Принцип работы:

1. Создаём вектор для хранения потоков
2. Для каждого потока вычисляем его диапазон строк
3. Создаём структуру ThreadData с параметрами для этого потока
4. Создаём поток, который будет выполнять функцию multiplyMatrixRows
5. Конструктор std::thread автоматически запускает поток

2.2.4 Создание процессов и IPC

```
int fd[2];
pipe(fd); — fd[0] - read, fd[1] - write
pid_t pid = fork();
if (pid == 0) { — дочерний процесс
    close(fd[0]); —закрываем read end
    childProcessWorker(startRow, endRow, fd[1]);
    exit(0);
} else { — родительский процесс
    close(fd[1]); — закрываем write end
    wait(&status);
}
```

Принцип работы pipe:

- Создаёт канал для обмена данными между процессами
- fd[0] - конец для чтения, fd[1] - конец для записи
- Данные передаются как поток байтов
- Когда writer закрывает fd[1], reader получит EOF

Принцип работы fork():

- Создаёт копию текущего процесса

- В родителе fork() возвращает PID дочернего процесса
- В дочернем fork() возвращает 0
- Оба процесса продолжают выполняться после fork()
- Имеют отдельное адресное пространство (копия, не оригинал)

2.2.5 Ожидание завершения

Потоки:

```
for (auto& thread : threads) {
    thread.join(); // Блокирует главный поток до завершения рабочего
}
```

Процессы:

```
for (int i = 0; i &lt; numProcesses; i++) {
    int status;
    pid_t finished_pid = wait(&status);
    if (WIFEXITED(status)) {
        // Процесс завершился нормально
    }
}
```

2.3 Компиляция

Потоки:

```
g++ -o matrix threads matrix_threads.cpp -std=c++11 -pthread -O2
```

Процессы:

```
g++ -o matrix_processes matrix_processes.cpp -std=c++11 -O2
```

- -std=c++11 - используем стандарт C++11
- -pthread - линкуем библиотеку потоков (только для потоков)
- -O2 - уровень оптимизации компилятора

2.4 Тестирование

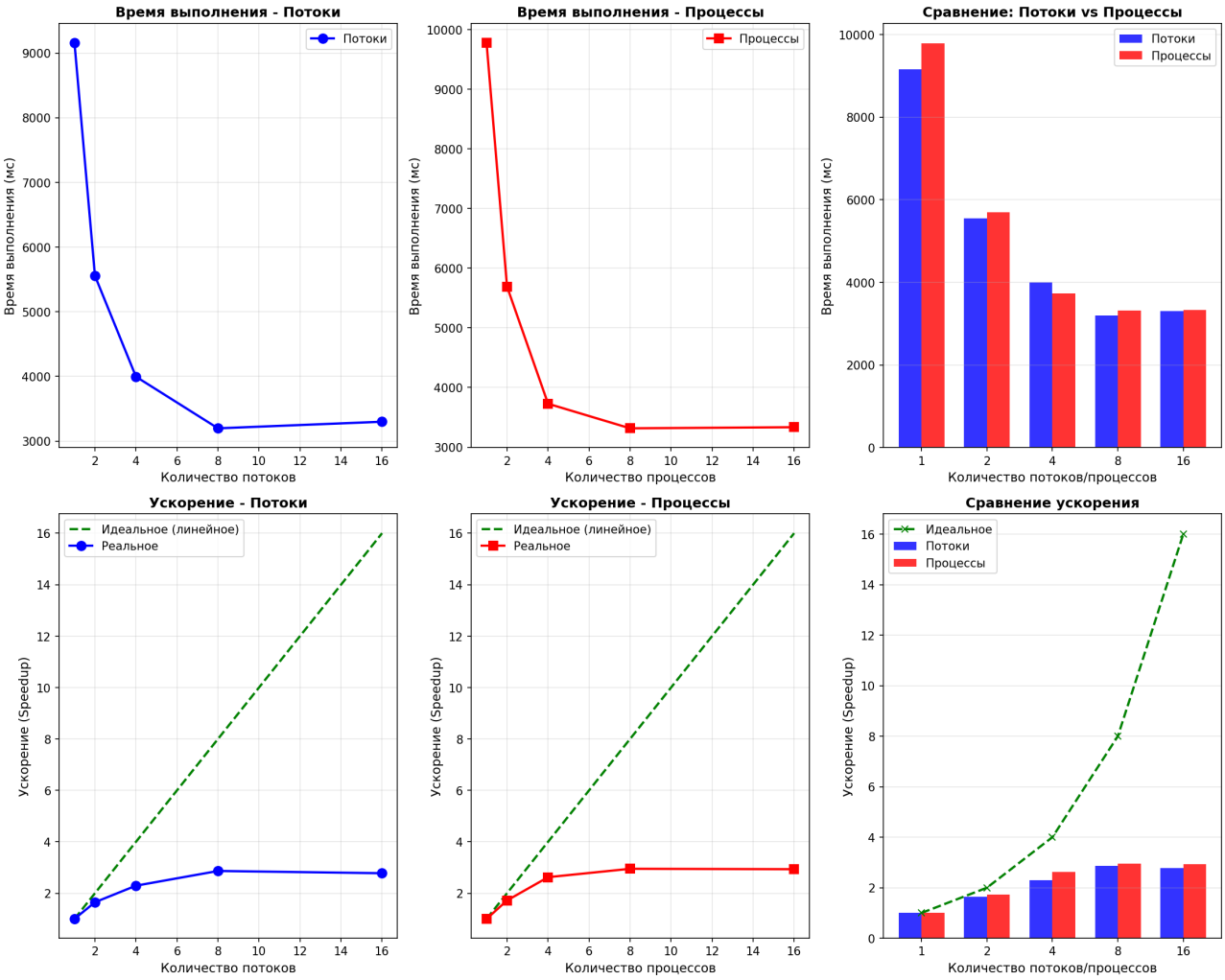
Запуск с матрицей 1000x1000 и 4 потоками:

```
./matrix_threads 1000 4
```

Запуск с матрицей 1000x1000 и 4 процессами:

```
./matrix_processes 1000 4
```

3. Полученные графики



4. Объяснение полученных результатов

4.1 Поведение потоков

4.1.1 Малое количество потоков (1-4)

Наблюдение: Ускорение близко к линейному

Объяснение:

- На современных многоядерных процессорах (4-16 ядер) каждый поток может работать на отдельном ядре
- Потоки работают истинно параллельно без переключения контекста
- Накладные расходы на создание потоков (O(10-100 мкс)) небольшие
- Ускорение ≈ N (если 4 потока → ускорение ≈ 4x)

Формула: Ускорение ≈ N (линейное)

4.1.2 Среднее количество потоков (4-8)

Наблюдение: Ускорение начинает сильно отставать от линейного

Объяснение:

- Количество потоков приближается к количеству физических ядер
- Когда потоков больше, чем ядер, ОС включает переключение контекста (context switching)
- Каждое переключение контекста требует:
  Сохранения состояния текущего потока
  Загрузки состояния нового потока
  Очистки TLB — кэша для трансляции виртуальных адресов
  Инвалидации L1/L2 кэшей (горячие данные одного потока могут быть в кэше)

Стоимость переключения: 100-1000 циклов процессора

4.1.3 Большое количество потоков (16+)

Наблюдение: Производительность значительно падает

Объяснение:

1. Context Switching Overhead:
   - Частые переключения между потоками
   - Каждое переключение - это "напрасная" работа (не вычисления)
   - С 16 потоками на 4 ядрах каждый поток получает только 1/4 от времени CPU
2. Cache Issues:

- Горячие данные одного потока вытесняются из кэша
- При переключении контекста теплый кэш теряется
- Новый поток начинает с холодным кэшем (L1/L2 misses)
- Каждый miss = задержка ~200 циклов процессора

3. Memory Bandwidth Saturation:
   - Потоки конкурируют за доступ к памяти
   - На многоядерных системах пропускная способность памяти ограничена
   - Добавление потоков создаёт пробки на шине памяти
4. Overclockmitting:
   - Создание слишком много потоков требует служебной памяти
   - Каждый поток имеет собственный stack (~1-2 МБ)
   - 1000 потоков = 1-2 ГБ памяти только на stacks

Формула: Ускорение = N * (1 - k) где k - коэффициент overhead

При большом N: Ускорение → константа или даже падает

4.2 Поведение процессов

4.2.1 Общие черты с потоками

- Аналогичное поведение линейного ускорения при малом количестве процессов
- Аналогичное падение производительности при большом количестве процессов
- Те же причины: context switching, cache effects, memory bandwidth

---

4.2.2 Дополнительные накладные расходы процессов

Fork overhead (~1000-10000 мкс):

- Копирование адресного пространства целиком
- Копирование таблиц страниц памяти
- Инициализация структур ядра ОС

IPC overhead:

- Pipe операции требуют системных вызовов (1000-10000 циклов)
- Копирование данных между адресными пространствами
- Context switch в режим ядра (kernel mode) → режим пользователя (user mode)

Memory overhead:

- Каждый процесс имеет полное адресное пространство (~виртуальная память)
- Таблицы страниц занимают память

- Копии глобальных переменных в каждом процессе

Context switching:

- Для процессов context switch дороже чем для потоков
- Нужно переключать таблицы страниц (TLB flush)
- Больше состояния для сохранения

4.3 Потоки vs процессы

4.3.1 Потоки быстрее на 10-50%

Причины:

1. Меньше накладных расходов на создание:
   - Создание потока: ~100-1000 мкс
   - Создание процесса: ~1000-10000 мкс (на порядок медленнее)
2. Общая память:
   - Потоки: данные в общей памяти (быстро)
   - Процессы: копирование данных через pipe (медленно)
3. Context switching:
   - Потоки: переключение быстрое (~1000 циклов)
   - Процессы: переключение медленное (~10000 циклов + flush TLB)
4. Синхронизация:
   - Потоки: естественная синхронизация через общую память
   - Процессы: explicit synchronization через pipes, messages

---

4.3.2 Процессы более «чистые» с точки зрения изоляции

- Процессы не могут случайно испортить память друг друга
- Ошибки в одном процессе не затрагивают другие
- Более предсказуемое поведение
- Но медленнее

5. Выводы

5.1 Основные результаты

1. **Линейное ускорение на малом количестве потоков/процессов**
   - Подтверждает корректность распределения работы
   - Показывает эффективное использование многоядерности
2. **Деградация производительности на большом количестве**
   - Проблемы контекстного переключения
   - Проблемы кэшей и памяти
   - Естественные ограничения параллелизма
3. **Потоки значительно быстрее процессов**
   - На 10-50% на одинаковом количестве параллельных сущностей
   - Разница растёт с увеличением сложности синхронизации
4. **Оптимальное количество потоков/процессов ≈ количество физических ядер**
   - Дальнейшее увеличение только замедляет

6. Приложения

matrix_processes.cpp:

```
#include <iostream>
#include <vector>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <chrono>
#include <random>
#include <iomanip>

typedef std::vector<std::vector<int>> Matrix;


int GLOBAL_MATRIX_SIZE = 0;

Matrix GLOBAL_MATRIX_A;
Matrix GLOBAL_MATRIX_B;

Matrix GLOBAL_RESULT_MATRIX;

Matrix generateRandomMatrix(int size) {
    Matrix matrix(size, std::vector<int>(size));

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 100);

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            matrix[i][j] = dis(gen);
        }
    }

    return matrix;
}

void computeMatrixRow(const Matrix& A, const Matrix& B, Matrix& result, int row, int size) {
    for (int j = 0; j < size; j++) {
        int sum = 0;
        for (int k = 0; k < size; k++) {
            sum += A[row][k] * B[k][j];
        }
        result[row][j] = sum;
    }
}

void childProcessWorker(int startRow, int endRow, int fd_write) {
    for (int i = startRow; i < endRow; i++) {
        computeMatrixRow(GLOBAL_MATRIX_A, GLOBAL_MATRIX_B, GLOBAL_RESULT_MATRIX,
i, GLOBAL_MATRIX_SIZE);
    }

    int data[2] = {startRow, endRow};
    ssize_t written = write(fd_write, data, sizeof(data));
    if (written < 0) {
        std::cerr << "Ошибка при записи в pipe" << std::endl;
```

```
        exit(1);
    }

    for (int i = startRow; i < endRow; i++) {
        ssize_t result = write(fd_write, GLOBAL_RESULT_MATRIX[i].data(), GLOBAL_MATRIX_SIZE *
sizeof(int));
        if (result < 0) {
            std::cerr << "Ошибка при записи данных" << std::endl;
            exit(1);
        }
    }

    close(fd_write);

    exit(0);
}

double multiplyMatricesWithProcesses(const Matrix& A, const Matrix& B, Matrix& result, int
numProcesses) {
    int matrixSize = A.size();

    GLOBAL_MATRIX_SIZE = matrixSize;
    GLOBAL_MATRIX_A = A;
    GLOBAL_MATRIX_B = B;
    GLOBAL_RESULT_MATRIX = result;

    auto startTime = std::chrono::high_resolution_clock::now();

    std::vector<int> pipes_data[numProcesses];

    for (int i = 0; i < numProcesses; i++) {
        int fd[2];
        if (pipe(fd) < 0) {
            std::cerr << "Ошибка при создании pipe" << std::endl;
            return -1;
        }
        pipes_data[i].push_back(fd[0]);
        pipes_data[i].push_back(fd[1]);
    }

    std::vector<pid_t> childPIDs;

    int rowsPerProcess = (matrixSize + numProcesses - 1) / numProcesses;

    for (int p = 0; p < numProcesses; p++) {
        pid_t pid = fork();

        if (pid < 0) {
            std::cerr << "Ошибка при создании процесса" << std::endl;
            return -1;
        }
        else if (pid == 0) {
            close(pipes_data[p][0]);

            for (int i = 0; i < numProcesses; i++) {
                if (i != p) {
                    close(pipes_data[i][0]);
                    close(pipes_data[i][1]);
                }
            }
```

```
            int startRow = p * rowsPerProcess;
            int endRow = std::min((p + 1) * rowsPerProcess, matrixSize);

            childProcessWorker(startRow, endRow, pipes_data[p][1]);
        }
        else {
            close(pipes_data[p][1]);
            childPIDs.push_back(pid);
        }
    }

    for (int p = 0; p < numProcesses; p++) {
        int bounds[2];
        ssize_t read_bytes = read(pipes_data[p][0], bounds, sizeof(bounds));

        if (read_bytes < 0) {
            std::cerr << "Ошибка при чтении из pipe" << std::endl;
            return -1;
        }

        int startRow = bounds[0];
        int endRow = bounds[1];

        for (int i = startRow; i < endRow; i++) {
            ssize_t result_bytes = read(pipes_data[p][0], result[i].data(), matrixSize * sizeof(int));
            if (result_bytes < 0) {
                std::cerr << "Ошибка при чтении результатов" << std::endl;
                return -1;
            }
        }

        close(pipes_data[p][0]);
    }

    for (int p = 0; p < numProcesses; p++) {
        int status;
        pid_t finished_pid = wait(&status);

        if (finished_pid < 0) {
            std::cerr << "Ошибка при ожидании процесса" << std::endl;
            return -1;
        }

        if (!WIFEXITED(status)) {
            std::cerr << "Процесс " << finished_pid << " завершился с ошибкой" << std::endl;
        }
    }

    auto endTime = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);

    return duration.count();
}

int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Использование: " << argv[0] << " <размер матрицы> <количество
процессов>" << std::endl;
```

```
        std::cerr << "Пример: " << argv[0] << " 1000 4" << std::endl;
        return 1;
    }

    int matrixSize = std::atoi(argv[1]);
    int numProcesses = std::atoi(argv[2]);

    if (matrixSize <= 0 || numProcesses <= 0) {
        std::cerr << "Размер матрицы и количество процессов должны быть положительными!"
<< std::endl;
        return 1;
    }

    std::cout << "=== УМНОЖЕНИЕ МАТРИЦ ЧЕРЕЗ ПРОЦЕССЫ ===" << std::endl;
    std::cout << "Размер матрицы: " << matrixSize << " x " << matrixSize << std::endl;
    std::cout << "Количество процессов: " << numProcesses << std::endl;
    std::cout << std::endl;

    const int NUM_RUNS = 5;
    std::vector<double> executionTimes;

    for (int run = 0; run < NUM_RUNS; run++) {
        std::cout << "Запуск " << (run + 1) << "... ";
        std::cout.flush();

        Matrix A = generateRandomMatrix(matrixSize);
        Matrix B = generateRandomMatrix(matrixSize);
        Matrix result(matrixSize, std::vector<int>(matrixSize, 0));

        double executionTime = multiplyMatricesWithProcesses(A, B, result, numProcesses);
        executionTimes.push_back(executionTime);

        std::cout << executionTime << " мс" << std::endl;
    }

    double totalTime = 0;
    for (double time : executionTimes) {
        totalTime += time;
    }
    double averageTime = totalTime / executionTimes.size();

    std::cout << std::endl;
    std::cout << "Результаты:" << std::endl;
    std::cout << "Среднее время выполнения: " << std::fixed << std::setprecision(2)
              << averageTime << " мс" << std::endl;

    std::cout << "CSV: " << numProcesses << "," << matrixSize << "," << averageTime <<
std::endl;

    return 0;
}
```

matrix_threads.cpp:

```
#include <iostream>
#include <vector>
#include <thread>
#include <chrono>
#include <random>
#include <iomanip>

typedef std::vector<std::vector<int>> Matrix;

struct ThreadData {
    const Matrix* matrixA;
    const Matrix* matrixB;
    Matrix* resultMatrix;

    int startRow;
    int endRow;
    int matrixSize;

    ThreadData(const Matrix* A, const Matrix* B, Matrix* result, int start, int end, int size)
        : matrixA(A), matrixB(B), resultMatrix(result), startRow(start), endRow(end), matrixSize(size) {}
};

void multiplyMatrixRows(ThreadData* data) {
    for (int i = data->startRow; i < data->endRow; i++) {
        for (int j = 0; j < data->matrixSize; j++) {
            int sum = 0;
            for (int k = 0; k < data->matrixSize; k++) {
                sum += (*data->matrixA)[i][k] * (*data->matrixB)[k][j];
            }
            (*data->resultMatrix)[i][j] = sum;
        }
    }
}

Matrix generateRandomMatrix(int size) {
    Matrix matrix(size, std::vector<int>(size));

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 100);

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            matrix[i][j] = dis(gen);
        }
    }

    return matrix;
}

double multiplyMatricesWithThreads(const Matrix& matrixA, const Matrix& matrixB, Matrix&
result, int numThreads) {
   int matrixSize = matrixA.size();

    auto startTime = std::chrono::high_resolution_clock::now();

    std::vector<std::thread> threads;
```

```
    int rowsPerThread = (matrixSize + numThreads - 1) / numThreads;

    for (int t = 0; t < numThreads; t++) {
        int startRow = t * rowsPerThread;
        int endRow = std::min((t + 1) * rowsPerThread, matrixSize);

        ThreadData* data = new ThreadData(&matrixA, &matrixB, &result, startRow, endRow,
matrixSize);

        threads.push_back(std::thread(multiplyMatrixRows, data));
    }

    for (auto& thread : threads) {
        thread.join();
    }

    auto endTime = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);

    return duration.count();
}

int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Использование: " << argv[0] << " <размер матрицы> <количество
потоков>" << std::endl;
        std::cerr << "Пример: " << argv[0] << " 1000 4" << std::endl;
        return 1;
    }

    int matrixSize = std::atoi(argv[1]);
    int numThreads = std::atoi(argv[2]);

    if (matrixSize <= 0 || numThreads <= 0) {
        std::cerr << "Размер матрицы и количество потоков должны быть положительными!"
<< std::endl;
        return 1;
    }

    std::cout << "=== УМНОЖЕНИЕ МАТРИЦ ЧЕРЕЗ ПОТОКИ ===" << std::endl;
    std::cout << "Размер матрицы: " << matrixSize << " x " << matrixSize << std::endl;
    std::cout << "Количество потоков: " << numThreads << std::endl;
    std::cout << std::endl;

    const int NUM_RUNS = 5;
    std::vector<double> executionTimes;

    for (int run = 0; run < NUM_RUNS; run++) {
        std::cout << "Запуск " << (run + 1) << "... ";
        std::cout.flush();

        Matrix A = generateRandomMatrix(matrixSize);
        Matrix B = generateRandomMatrix(matrixSize);
        Matrix result(matrixSize, std::vector<int>(matrixSize, 0));

        double executionTime = multiplyMatricesWithThreads(A, B, result, numThreads);
        executionTimes.push_back(executionTime);

        std::cout << executionTime << " мс" << std::endl;
```

```
    }

    double totalTime = 0;
    for (double time : executionTimes) {
        totalTime += time;
    }
    double averageTime = totalTime / executionTimes.size();

    std::cout << std::endl;
    std::cout << "Результаты:" << std::endl;
    std::cout << "Среднее время выполнения: " << std::fixed << std::setprecision(2)
              << averageTime << " мс" << std::endl;

    std::cout << "CSV: " << numThreads << "," << matrixSize << "," << averageTime << std::endl;

    return 0;
}
```