

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ

Московский Авиационный Институт
(Национальный Исследовательский Университет)

Институт №8 "Компьютерные науки и прикладная математика"
Кафедра 806 "Вычислительная математика и программирование"

Лабораторная работа №2
По курсу «Операционные системы»

Студент: Махмутов Д. И.

Группа: М8О-203Б-23

Вариант: 2

Преподаватель: Миронов Е. С.

Дата: _____

Оценка: _____

Подпись: _____

Москва, 2024

Содержание

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Сборка программы
7. Демонстрация работы программы
8. Выводы

Репозиторий

<https://github.com/mxdesta/osLabs/tree/main/lab3>

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.

Вариант задания:

2.Отсортировать массив целых чисел при помощи параллельного алгоритма быстрой сортировки

Общие сведения о программе

Программа представляет собой набор инструментов для параллельных вычислений, включающий в себя:

Сложение матриц — функция SumMatrices, которая позволяет складывать две матрицы с использованием многопоточности.

Быстрая сортировка (QuickSort) — класс QuickSort, реализующий параллельную версию алгоритма быстрой сортировки с использованием пула потоков.

Пул потоков (ThreadPool) — класс ThreadPool, который управляет созданием и выполнением задач в многопоточной среде.

Программа использует стандартную библиотеку C++ и POSIX-потоки (pthread) для реализации многопоточности.

Общий метод и алгоритм решения

Быстрая сортировка (QuickSort)

Класс QuickSort реализует параллельную версию алгоритма быстрой сортировки. Основной метод заключается в рекурсивном разделении массива на подмассивы с использованием опорного элемента (pivot). Если размер подмассива превышает 1000 элементов, рекурсивные вызовы сортировки выполняются в отдельных потоках через пул потоков. Для небольших подмассивов (размером меньше или равным 1000 элементов) сортировка выполняется в текущем потоке. После завершения всех задач в пуле потоков массив считается отсортированным. Этот подход позволяет эффективно использовать многопоточность для ускорения сортировки больших массивов.

Пул потоков (ThreadPool)

Класс ThreadPool управляет созданием и выполнением задач в многопоточной среде. При создании пула инициализируется фиксированное количество потоков, которые ожидают появления задач в очереди. Когда задача добавляется в очередь, один из потоков забирает ее и выполняет. После выполнения задачи поток уведомляет пул о завершении и продолжает ожидать новые задачи. Синхронизация потоков обеспечивается с использованием мьютексов и условных переменных. Пул потоков позволяет эффективно распределять задачи между потоками и управлять их выполнением.

Исходный код

quicksort.h:

```
#ifndef QUICKSORT_H
#define QUICKSORT_H

#include <vector>
#include "thread_pool.h"

class QuickSort {
public:
    explicit QuickSort(size_t maxThreads);
    void sort(std::vector<int>& arr);

private:
    ThreadPool threadPool;
```

```

    void parallelQuickSort(std::vector<int>& arr, int left, int right);

    int partition(std::vector<int>& arr, int left, int right);

};

#endif

quicksort.cpp:

#include "quicksort.h"

QuickSort::QuickSort(size_t maxThreads) : threadPool(maxThreads) {}

void QuickSort::sort(std::vector<int>& arr) {
    parallelQuickSort(arr, 0, arr.size() - 1);
    threadPool.waitForCompletion();
}

void QuickSort::parallelQuickSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int pivotIndex = partition(arr, left, right);
        int pivotIndexLeft = pivotIndex - 1;
        int pivotIndexRight = pivotIndex + 1;

        if ((right - left) > 1000) {
            threadPool.enqueueTask([this, &arr, left, pivotIndexLeft]() {
                parallelQuickSort(arr, left, pivotIndexLeft);
            });
            threadPool.enqueueTask([this, &arr, pivotIndexRight, right]() {
                parallelQuickSort(arr, pivotIndexRight, right);
            });
        }
    }
}

```

```

    } else {
        parallelQuickSort(arr, left, pivotIndexLeft);
        parallelQuickSort(arr, pivotIndexRight, right);
    }
}

}

}

int QuickSort::partition(std::vector<int>& arr, int left, int right) {
    int pivot = arr[right];
    int i = left - 1;
    for (int j = left; j <= right - 1; ++j) {
        if (arr[j] < pivot) {
            ++i;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[right]);
    return i + 1;
}

```

thread_pool.h

```

#ifndef THREAD_POOL_H
#define THREAD_POOL_H

#include <pthread.h>
#include <queue>
#include <vector>
#include <functional>

```

```

class ThreadPool {
public:
    explicit ThreadPool(size_t maxThreads);
    ~ThreadPool();

    void enqueueTask(std::function<void()> task);
    void waitForCompletion();

private:
    std::vector<pthread_t> workers;
    std::queue<std::function<void()>> tasks;
    pthread_mutex_t queueMutex;
    pthread_cond_t condition;
    size_t activeTasks;
    bool stop;

    static void* workerThread(void* arg);
    void processTasks();
    void taskCompleted();
};

#endif

```

thread_pool.cpp

```

#include "thread_pool.h"
#include <stdexcept>

```

```

ThreadPool::ThreadPool(size_t maxThreads) : activeTasks(0), stop(false) {
    pthread_mutex_init(&queueMutex, nullptr);

```

```

pthread_cond_init(&condition, nullptr);

for (size_t i = 0; i < maxThreads; ++i) {
    pthread_t thread;
    if (pthread_create(&thread, nullptr, workerThread, this) != 0) {
        throw std::runtime_error("Failed to create thread");
    }
    workers.push_back(thread);
}
}

```

```

ThreadPool::~ThreadPool() {
    {
        pthread_mutex_lock(&queueMutex);
        stop = true;
        pthread_mutex_unlock(&queueMutex);
        pthread_cond_broadcast(&condition);
    }

    for (pthread_t &worker : workers) {
        pthread_join(worker, nullptr);
    }

    pthread_mutex_destroy(&queueMutex);
    pthread_cond_destroy(&condition);
}

```

```

void ThreadPool::enqueueTask(std::function<void()> task) {

```



```

pthread_mutex_lock(&queueMutex);
tasks.push(std::move(task));
++activeTasks;
pthread_mutex_unlock(&queueMutex);
pthread_cond_signal(&condition);
}

void ThreadPool::taskCompleted() {
    pthread_mutex_lock(&queueMutex);
    --activeTasks;
    if (activeTasks == 0 && tasks.empty()) {
        pthread_cond_broadcast(&condition);
    }
    pthread_mutex_unlock(&queueMutex);
}

void ThreadPool::waitForCompletion() {
    pthread_mutex_lock(&queueMutex);
    while (activeTasks > 0 || !tasks.empty()) {
        pthread_cond_wait(&condition, &queueMutex);
    }
    pthread_mutex_unlock(&queueMutex);
}

void* ThreadPool::workerThread(void* arg) {
    auto* pool = static_cast<ThreadPool*>(arg);
    pool->processTasks();
    return nullptr;
}

```

```

void ThreadPool::processTasks() {
    while (true) {
        std::function<void()> task;

        {
            pthread_mutex_lock(&queueMutex);
            while (!stop && tasks.empty()) {
                pthread_cond_wait(&condition, &queueMutex);
            }
            if (stop && tasks.empty()) {
                pthread_mutex_unlock(&queueMutex);
                return;
            }
            task = std::move(tasks.front());
            tasks.pop();
            pthread_mutex_unlock(&queueMutex);
        }
        task();
        taskCompleted();
    }
}

```

lab3.cpp

```
#include "lab3.h"
```

```
#include "utils.h"
```

```
#include <thread>
```

```
namespace {
```

```

void SumGivenRows(const TMatrix& lhs, const TMatrix& rhs, TMatrix& result, int firstRow,
int lastRow) {

    int m = isize(lhs);

    for(int i = firstRow; i < lastRow; ++i) {

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

            result[i][j] = lhs[i][j] + rhs[i][j];

        }

    }

}

```

```

TMatrix SumMatrices(const TMatrix& lhs, const TMatrix& rhs, int threadCount) {

    TMatrix result(lhs.size(), std::vector<int>(lhs[0].size()));

    if(threadCount > 1) {

        int actualThreads = std::min(threadCount, isize(result));

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

        threads.reserve(actualThreads);

        int rowsPerThread = isize(result) / actualThreads;

        for(int i = 0; i < isize(result); i += rowsPerThread) {

            if(i + rowsPerThread >= isize(result)) {

                threads.emplace_back(SumGivenRows, std::ref(lhs), std::ref(rhs), std::ref(result), i,
isize(result));

            } else {

                threads.emplace_back(SumGivenRows, std::ref(lhs), std::ref(rhs), std::ref(result),
i, i + rowsPerThread);

            }

        }

    }

}

```

```

    }

    for(auto& thread : threads) {
        thread.join();
    }
} else {
    SumGivenRows(lhs, rhs, result, 0, lhs.size());
}

return result;
}

lab3.h

#ifndef OS_LABS_LAB3_H
#define OS_LABS_LAB3_H

#include <vector>

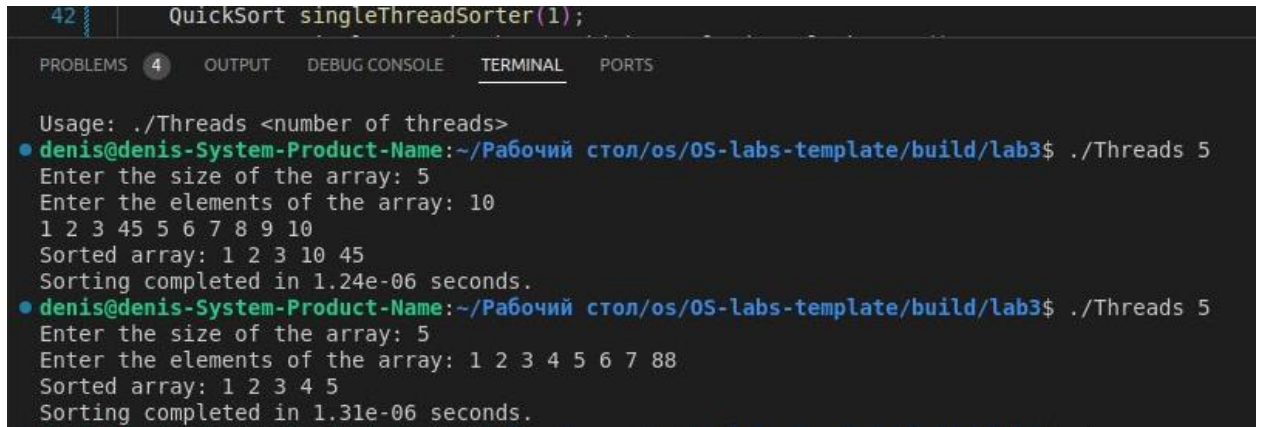
using TMatrix = std::vector<std::vector<int>>>;

TMatrix SumMatrices(const TMatrix& lhs, const TMatrix& rhs, int threadCount);

#endif //OS_LABS_LAB3_H

```

Демонстрация работы программы



```
42 | QuickSort singleThreadSorter(1);
PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS

Usage: ./Threads <number of threads>
• denis@denis-System-Product-Name:~/Рабочий стол/os/OS-labs-template/build/lab3$ ./Threads 5
Enter the size of the array: 10
Enter the elements of the array: 1 2 3 45 5 6 7 8 9 10
Sorted array: 1 2 3 10 45
Sorting completed in 1.24e-06 seconds.
• denis@denis-System-Product-Name:~/Рабочий стол/os/OS-labs-template/build/lab3$ ./Threads 5
Enter the size of the array: 5
Enter the elements of the array: 1 2 3 4 5 6 7 88
Sorted array: 1 2 3 4 5
Sorting completed in 1.31e-06 seconds.
```

Вывод

В целом, программа демонстрирует эффективное использование многопоточности для решения вычислительно сложных задач. Она может быть полезна в приложениях, где требуется высокая производительность и параллельная обработка данных.