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: Останина Анна Андреевна

Группа: М8О-208Б-22

Вариант: 2

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2024

**Содержание**

1. Репозиторий
2. Цель работы
3. Задание
4. Описание работы программы
5. Исходный код
6. Консоль
7. Запуск тестов
8. Выводы

### Репозиторий

<https://github.com/Imariiii/os_labs>

### Цель работы

### Приобретение практических навыков в управлении потоками в ОС и обеспечении синхронизации между потоками

### Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработке 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.

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

### Описание работы программы

Программа компилируется из файлов sort.cpp и main.cpp. Также имеется заголовочный файл sort.h и файл с тестами lab2\_test.cpp. В программе работы были использованы следующие системные вызовы:

* pthread\_create() - создание нового потока,
* pthread\_join() - ожидание завершения исполнения потока.

**Исходный код**

sort.h

#**pragma once**

#include <vector>

**class** ParallelSort {

**using** ll = **long** **long**;

**public**:

**static** **void** QuickSort(std::vector<ll>& vec, **unsigned** **int** threadsNum);

**private**:

**struct** QuickSortArgs {

std::vector<ll>\* vec;

std::size\_t left;

std::size\_t right;

**unsigned** **int** id;

std::size\_t numsPerThread;

};

**static** **void**\* QuickSort(**void**\* ptr);

**static** **void** QuickSort(std::vector<ll>& vec, std::size\_t left,

std::size\_t right);

**static** std::size\_t Partition(std::vector<ll>& vec, std::size\_t left,

std::size\_t right);

};

sort.cpp

#include "sort.h"

#include <pthread.h>

#include <algorithm>

#include <iostream>

#include <vector>

pthread\_barrier\_t barrier;

std::size\_t ParallelSort::Partition(std::vector<ll>& vec, std::size\_t left,

std::size\_t right) {

ll elem = vec[(left + right) / 2];

**while** (left <= right) {

**while** (vec[left] < elem) {

++left;

}

**while** (vec[right] > elem) {

--right;

}

**if** (left >= right) {

**break**;

}

std::swap(vec[left++], vec[right--]);

}

**return** right;

}

**void** ParallelSort::QuickSort(std::vector<ll>& vec, std::size\_t left,

std::size\_t right) {

**if** (left < right) {

std::size\_t pivot = Partition(vec, left, right);

QuickSort(vec, left, pivot);

QuickSort(vec, pivot + 1, right);

}

}

**void**\* ParallelSort::QuickSort(**void**\* ptr) {

**auto**\* args = **static\_cast**<QuickSortArgs\*>(ptr);

**auto**& vec = \*args->vec;

QuickSort(vec, args->left, args->right);

pthread\_barrier\_wait(&barrier);

std::size\_t step = args->numsPerThread;

**while** (step < vec.size()) {

**if** (args->id % 2 == 0) {

args->id /= 2;

std::size\_t left = args->left;

std::size\_t mid =

std::min<std::size\_t>(args->left + step, vec.size());

std::size\_t right =

std::min<std::size\_t>(args->left + 2 \* step, vec.size());

std::inplace\_merge(vec.begin() + left, vec.begin() + mid,

vec.begin() + right);

}

step \*= 2;

pthread\_barrier\_wait(&barrier);

}

**return** **nullptr**;

}

**void** ParallelSort::QuickSort(std::vector<ll>& vec, **unsigned** **int** threadsNum) {

std::size\_t numsPerThread;

**if** (vec.size() < threadsNum) {

numsPerThread = vec.size();

threadsNum = 1;

} **else** {

numsPerThread = vec.size() / threadsNum;

}

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

std::vector<QuickSortArgs> args(threadsNum);

pthread\_barrier\_init(&barrier, **nullptr**, threadsNum);

**for** (**unsigned** **int** i = 0; i < threadsNum; ++i) {

std::size\_t begin = i \* numsPerThread;

std::size\_t end =

(i == threadsNum - 1) ? vec.size() - 1 : begin + numsPerThread - 1;

args[i] = {&vec, begin, end, i, numsPerThread};

pthread\_create(&threads[i], **nullptr**, &QuickSort, &args[i]);

}

**for** (**unsigned** **int** i = 0; i < threadsNum; ++i) {

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

}

pthread\_barrier\_destroy(&barrier);

}

main.cpp

#include <algorithm>

#include <chrono>

#include <iostream>

#include <limits>

#include <random>

#include <vector>

#include "sort.h"

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

**if** (argc != 2) {

std::cerr << "usage: sort THREADS\n";

std::exit(EXIT\_SUCCESS);

}

std::random\_device rndDevice;

std::mt19937 mersenneEngine{rndDevice()};

std::uniform\_int\_distribution<**long** **long**> dist{

std::numeric\_limits<**long** **long**>::min(),

std::numeric\_limits<**long** **long**>::max()};

**auto** gen = [&dist, &mersenneEngine]() { **return** dist(mersenneEngine); };

std::vector<**long** **long**> vec(1e7);

std::generate(std::begin(vec), std::end(vec), gen);

**for** (**unsigned** **int** i = 1;

i < **static\_cast**<**unsigned** **int**>(std::stoi(argv[1])+1); ++i) {

**auto** vecCopy = vec;

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

ParallelSort::QuickSort(vecCopy, i);

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

**auto** duration =

std::chrono::duration\_cast<std::chrono::microseconds>(stop - start);

**if** (std::is\_sorted(vecCopy.begin(), vecCopy.end())) {

std::cout << "Sorted on " << i << " threads in " << duration.count()

<< "ms" << std::endl;

} **else** {

std::cout << "Something wrong" << std::endl;

}

}

**return** 0;

}

lab2\_test.cpp

#include <gtest/gtest.h>

#include <algorithm>

#include <chrono>

#include <iostream>

#include <limits>

#include <random>

#include <vector>

#include <thread>

#include "sort.h"

constexpr **long** **long** LEN = 1e6;

std::vector<**long** **long**> RandomVec(std::size\_t len) {

std::random\_device rndDevice;

std::mt19937 mersenneEngine{rndDevice()};

std::uniform\_int\_distribution<**long** **long**> dist{

std::numeric\_limits<**long** **long**>::min(),

std::numeric\_limits<**long** **long**>::max()};

**auto** gen = [&dist, &mersenneEngine]() { **return** dist(mersenneEngine); };

std::vector<**long** **long**> vec(len);

std::generate(std::begin(vec), std::end(vec), gen);

**return** vec;

}

TEST(SecondLabTests, SimpleTest) {

std::vector<**long** **long**> vec = {1, 4, 3, -13, 4};

**const** **char** \*threadsNumStr = std::getenv("THREADS\_NUM");

ASSERT\_TRUE(threadsNumStr);

**const** **unsigned** **int** threadsNum = std::stoi(threadsNumStr);

**for** (**unsigned** **int** i = 1; i < threadsNum+1; ++i) {

ParallelSort::QuickSort(vec, i);

EXPECT\_TRUE(std::is\_sorted(vec.begin(), vec.end()));

}

}

TEST(SecondLabTests, ReliabilityTest) {

**auto** vec = RandomVec(LEN);

ParallelSort::QuickSort(vec, 5);

EXPECT\_TRUE(std::is\_sorted(vec.begin(), vec.end()));

}

TEST(SecondLabTests, MultithreadTest) {

**auto** vec = RandomVec(LEN);

**const** **char** \*threadsNumStr = std::getenv("THREADS\_NUM");

ASSERT\_TRUE(threadsNumStr);

**const** **unsigned** **int** threadsNum = std::stoi(threadsNumStr);

**for** (**unsigned** **int** i = 1; i < threadsNum+1; ++i) {

**auto** vecCopy = vec;

ParallelSort::QuickSort(vecCopy, i);

EXPECT\_TRUE(std::is\_sorted(vecCopy.begin(), vecCopy.end()));

}

}

TEST(SecondLabTests, EfficiencyTest) {

**auto** vec = RandomVec(1e7);

**const** **char** \*threadsNumStr = getenv("THREADS\_NUM");

ASSERT\_TRUE(threadsNumStr);

**const** **unsigned** **int** threadsNum = std::stoi(threadsNumStr);

int64\_t prevDuration = INT64\_MAX;

**for** (**unsigned** **int** i = 0; i < 2; i++) {

**unsigned** **int** th = (i == 0) ? 1 : threadsNum;

**auto** vecCopy = vec;

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

ParallelSort::QuickSort(vecCopy, th);

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

**auto** duration =

std::chrono::duration\_cast<std::chrono::microseconds>(stop - start)

.count();

EXPECT\_TRUE(std::is\_sorted(vecCopy.begin(), vecCopy.end()));

EXPECT\_LT(duration, prevDuration);

prevDuration = duration;

**if** (threadsNum == 1) {

**break**;

}

}

}

Консоль

anna@anna-**virtual**-machine:~/labs\_3sem/os\_labs/build/lab2$ ./sort 4

Sorted on 1 threads in 5278872ms

Sorted on 2 threads in 3486179ms

Sorted on 3 threads in 2577430ms

Sorted on 4 threads in 2902248ms

### Запуск тестов

anna@anna-**virtual**-machine:~/labs\_3sem/os\_labs/build/tests$ THREADS\_NUM=4 ./lab2\_test

Running main() from /home/anna/labs\_3sem/os\_labs/build/\_deps/googletest-src/googletest/src/gtest\_main.cc

[==========] Running 4 tests from 1 test suite.

[----------] Global test environment set-up.

[----------] 4 tests from SecondLabTests

[ RUN ] SecondLabTests.SimpleTest

[ OK ] SecondLabTests.SimpleTest (36 ms)

[ RUN ] SecondLabTests.ReliabilityTest

[ OK ] SecondLabTests.ReliabilityTest (576 ms)

[ RUN ] SecondLabTests.MultithreadTest

[ OK ] SecondLabTests.MultithreadTest (1593 ms)

[ RUN ] SecondLabTests.EfficiencyTest

[ OK ] SecondLabTests.EfficiencyTest (9266 ms)

[----------] 4 tests from SecondLabTests (11477 ms total)

[----------] Global test environment tear-down

[==========] 4 tests from 1 test suite ran. (11479 ms total)

[ PASSED ] 4 tests.

Выводы

В ходе выполнения лабораторной работы я изучила механизм работы многопоточности с использованием библиотеки pthreads в операционной системе Linux. Было выявлено, что ускорение алгоритма и его эффективность зависят от размера входного массива и количества доступных потоков для параллельной обработки. При увеличении размера массива или количества потоков скорость выполнения алгоритма увеличивается, однако при достижении определенного количества потоков или размера массива скорость может перестать увеличиваться из-за расходов на управление потоками и синхронизацию данных.

Также были изучены средства синхронизации такие, как семафор, мьютекс, барьер, условные переменные, и основные проблемы многопоточности.