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

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.