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: Немкова Анастасия Романовна

Группа: М8О-208Б-22

Преподаватель: Миронов Евгений Сергеевич

Вариант: 1

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2023

Содержание

Репозиторий

`https://github.com/anastasia-nemkova/OS_labs`

Постановка задачи

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 1: Отсортировать массив целых чисел при помощи битонической сортировки

Общие сведения о программе

Программа компилируется из файлов bitonic-sort.cpp, bitonic-sort.hpp и main.cpp. Также имеются с 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

Общий метод и алгоритм решения

Алгоритм битонической сортировки состоит из двух основных этапов: сортировки и слияния. На этапе сортировки считанный массив в функции parallelBitonicSort разбивается на две части, каждая из которых сортируется в отдельном потоке по возрастанию или убыванию(в зависимости от заданного напрвления). После этого происходит слияние отсортированных частей в один массив, который выводится на экран.

Исходный код

bitonic-sort.hpp

```hpp
#pragma once

#include <iostream>
#include <vector>
#include <pthread.h>

struct Data {
    std::vector<int>& a;
    int low; // нижний индекс массива
    int cnt; //количество элементов
    int dir; // 1 — по возрастанию; 0 — по убыванию
    int maxThreads;
};

void compAndSwap(std::vector<int>& a, int i, int j, int dir);
void bitonicMerge(std::vector<int>& a, int low, int cnt, int dir);
void bitonicSort(std::vector<int>& a, int low, int cnt, int dir) ;
void* parallelBitonicSort(void* arg);
void sort(std::vector<int> &a, int up, int maxThreads);
```

bitonic-sort.cpp

```cpp
#include "bitonic-sort.hpp"


void compAndSwap(std::vector<int> &a, int i, int j, int dir) {
    if (dir == (a[i] > a[j])) {
        std::swap(a[i], a[j]);
    }
}

void bitonicMerge(std::vector<int> &a, int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        for (int i = low; i < low + k; i++)
            compAndSwap(a, i, i + k, dir);
        bitonicMerge(a, low, k, dir);
        bitonicMerge(a, low + k, k, dir);
    }
}

void bitonicSort(std::vector<int> &a, int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        bitonicSort(a, low, k, 1);
        bitonicSort(a, low + k, k, 0);
        bitonicMerge(a, low, cnt, dir);
    }
}

void* parallelBitonicSort(void* arg) {
    Data* data = static_cast<Data*>(arg);
    std::vector<int>& a = data->a;
    int low = data->low;
    int cnt = data->cnt;
    int dir = data->dir;
    int maxThreads = data->maxThreads;

    if (cnt > 1) {
        if (maxThreads == 1 || cnt <= 2) {
            bitonicSort(a, low, cnt, dir);
        } else {
            int k = cnt / 2;
            pthread_t t1, t2;
            Data data1 = {a, low, k, 1, maxThreads / 2};
            Data data2 = {a, low + k, k, 0, maxThreads / 2};
```

```
            pthread_create(&t1, nullptr, parallelBitonicSort, &data1);
            pthread_create(&t2, nullptr, parallelBitonicSort, &data2);
            pthread_join(t1, nullptr);
            pthread_join(t2, nullptr);
            bitonicMerge(a, low, cnt, dir);
        }
    }
    return nullptr;
}

void sort(std::vector<int> &a, int up, int maxThreads) {
    int n = a.size();
    Data data = {a, 0, n, up, maxThreads};
    parallelBitonicSort(&data);
}
```

main.cpp

```
#include "bitonic-sort.hpp"


int main(int argc, char const *argv[]) {
    if (argc != 2){
        std::cout << "wrong arguments" << std::endl;
        return -1;
    }
    int maxThreads = std::atoi(argv[1]);

    int n;
    std::cout << "Size of arrive: ";
    std::cin >> n;
    std::vector<int> arr(n);

    std::cout << "Element of arrive: ";
    for(int i = 0; i < n; ++i){
        std::cin >> arr[i];
    }
    // up = 1 - сортировка по возрастанию
    // up = 0 - сортировка по убыванию
    int up = 1;
    sort(arr, up, maxThreads);

    std::cout << "Result:\n";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }

    return 0;
}
```

lab2_test.cpp

```
#include <gtest/gtest.h>

#include <bitonic-sort.hpp>
#include <cmath>
#include <chrono>

std::vector<int> GenerateArray(int n) {
    std::vector<int> result(n);
    srand(static_cast<unsigned>(time(nullptr)));
    for(int i = 0; i < n; ++i) {
        result[i] = std::rand() % 100;
    }
    return result;
}


TEST(SecondLabTests, SingleThreadSort) {
    std::vector<int> a = {4, 5, 3, 1};
```

```
19      std::vector<int> a_sorted = a;
20      std::sort(a_sorted.begin(), a_sorted.end());
21      sort(a, 1, 1);
22      EXPECT_EQ(a, a_sorted);
23
24      std::vector<int> b = {8, 6, 1, 4, 2, 9, 1, 4};
25      std::vector<int> b_sorted = b;
26      std::sort(b_sorted.begin(), b_sorted.end());
27      sort(b, 1, 1);
28      EXPECT_EQ(b,  b_sorted);
29
30      std::vector<int> c = {8, 6, 1, 4, 2, 9, 1, 4};
31      std::vector<int> c_sorted = c;
32      std::sort(c_sorted.begin(), c_sorted.end(), std::greater<int>());
33      sort(c, 0, 1);
34      EXPECT_EQ(c, c_sorted);
35  }
36
37  TEST(SecondLabTest, MultithreadedSort) {
38      auto performTestForGivenSize = [](int n, int maxThreadCount) {
39          auto result = GenerateArray(n);
40          auto sorted = result;
41          sort(result, 1, maxThreadCount);
42          std::sort(sorted.begin(), sorted.end());
43          EXPECT_EQ(sorted, result);
44      };
45
46      performTestForGivenSize(4, 2);
47      performTestForGivenSize(8, 4);
48      performTestForGivenSize(1024, 6);
49      performTestForGivenSize(32768, 10);
50  }
51
52  TEST(SecondLabTest, TimeSort) {
53      auto getAvgTime = [](int threadCount) {
54          int size_arr = std::pow(2, 23);
55          auto arr = GenerateArray(size_arr);
56
57          auto begin = std::chrono::high_resolution_clock::now();
58          sort(arr, 1, threadCount);
59          auto end = std::chrono::high_resolution_clock::now();
60          double result = std::chrono::duration_cast<std::chrono::milliseconds>(end
        - begin).count();
61          return result;
62      };
63
64      auto singleThread = getAvgTime(1);
65      auto multiThread = getAvgTime(4);
66
67      std::cout << "Avg time for 1 thread: " << singleThread << '\n';
68      std::cout << "Avg time for 4 threads: " << multiThread << '\n';
69
70      EXPECT_GE(singleThread, multiThread);
71  }
72
73  int main(int argc, char **argv) {
74      testing::InitGoogleTest(&argc, argv);
75      return RUN_ALL_TESTS();
76  }
```

Демонстрация работы программы

```
arnemkova@LAPTOP-TA2RV74U:~/OS_labs/build$ ./tests/lab2_test
[==========] Running 3 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 1 test from SecondLabTests
[ RUN      ] SecondLabTests.SingleThreadSort
[       OK ] SecondLabTests.SingleThreadSort (0 ms)
[----------] 1 test from SecondLabTests (0 ms total)

[----------] 2 tests from SecondLabTest
[ RUN      ] SecondLabTest.MultithreadedSort
[       OK ] SecondLabTest.MultithreadedSort (16 ms)
[ RUN      ] SecondLabTest.TimeSort
Avg time for 1 thread: 14555
Avg time for 4 threads: 6343
[       OK ] SecondLabTest.TimeSort (21103 ms)
[----------] 2 tests from SecondLabTest (21120 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 2 test suites ran. (21120 ms total)
[  PASSED  ] 3 tests.
```

Вывод

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, баррьер, условные переменные, и основные проблемы многопоточности.