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»**

Группа: М8О-203Б-23

Студент: Никитцев А.В.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 25.12.24

Москва, 2024

**Цель работы**

Целью является приобретение практических навыков в:

* Управление потоками в ОС
* Обеспечение синхронизации между потоками

**Постановка задачи (Вариант 4)**

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.

Отсортировать массив целых чисел при помощи TimSort

**Код**

**Timsort.cpp**

#include <iostream>  
#include <vector>  
#include <stack>  
#include <queue>  
#include <tuple>  
#include <semaphore>  
  
const int THREAD\_SIZE = 4;  
  
sem\_t sem;  
  
struct InsertParams {  
 long\* values;  
 size\_t start;  
 size\_t end;  
};  
  
struct MergeParams {  
 long\* values;  
 size\_t start1;  
 size\_t end1;  
 size\_t start2;  
 size\_t end2;  
};  
  
namespace timsort {  
  
 size\_t getSplits(std::queue<std::pair<size\_t, size\_t>>\* splits, size\_t size, size\_t threads = 8) {  
 size\_t curI = 0;  
 while (curI < size){  
 auto next = std::min(size, curI + size / threads );  
 splits->emplace(curI, next);  
 curI = next;  
 }  
 return size / threads;  
 }  
  
 void move(std::vector<long>& values, long start, long end){  
 while (start < end){  
 values[end] = values[end - 1];  
 end--;  
 }  
 }  
  
 void insertionSort(long\* values, size\_t start, size\_t end) {  
 auto v = std::vector<long>(end - start);  
 auto sz = 0;  
  
 for (auto cur = start; cur < end; ++cur){  
 auto i = 0;  
 while (i < sz && values[cur] >= v[i]){  
 i++;  
 }  
  
 if (i == sz){  
 v[i] = values[cur];  
 sz++;  
 continue;  
 }  
  
 move(v, i, sz);  
 sz++;  
 v[i] = values[cur];  
 }  
  
 for (auto i = start; i < end; ++i){  
 values[i] = v[i - start];  
 }  
 }  
  
 void\* insertSortThread(void\* params){  
 auto p = static\_cast<InsertParams\*>(params);  
 insertionSort(p->values, p->start, p->end);  
 std::free(params);  
 return nullptr;  
 }  
  
 void mergeSort(long\* values, size\_t start1, size\_t end1, size\_t start2, size\_t end2){  
 auto v = std::vector<long>(end2 - start1);  
  
 auto i = start1;  
 auto j = start2;  
 auto k = 0;  
  
 while (i < end1 && j < end2){  
 if(values[i] < values[j]) {  
 v[k] = values[i];  
 ++i;  
 }  
 else {  
 v[k] = values[j];  
 ++j;  
 }  
 ++k;  
 }  
  
 while (i < end1){  
 v[k] = values[i];  
 ++i;  
 ++k;  
 }  
  
 while (j < end2){  
 v[k] = values[j];  
 ++j;  
 ++k;  
 }  
  
 for (auto s = start1; s < end2; ++s){  
 values[s] = v[s - start1];  
 }  
 }  
  
 void\* mergeSortThread(void\* params){  
 auto p = static\_cast<MergeParams\*>(params);  
 mergeSort(p->values, p->start1, p->end1, p->start2, p->end2);  
 std::free(params);  
 return nullptr;  
 }  
  
 void sort(long \*values, size\_t size) {  
 std::queue<std::pair<size\_t, size\_t>> splits;  
  
 auto mergeSize = getSplits(&splits, size);  
  
 while (!splits.empty()){  
 auto p = splits.front();  
 splits.pop();  
 insertionSort(values, p.first, p.second);  
 }  
  
 while (mergeSize <= size){  
 for (size\_t start = 0; start < size; start += 2 \* mergeSize) {  
 size\_t mid = std::min(start + mergeSize, size);  
 size\_t end = std::min(start + 2 \* mergeSize, size);  
  
 if (mid < end) {  
 mergeSort(values, start, mid, mid, end);  
 }  
 }  
  
 mergeSize \*= 2;  
 }  
 }  
  
 void sortAsync(long \*values, size\_t size, size\_t thr) {  
 std::queue<std::pair<size\_t, size\_t>> splits;  
 sem\_init(&sem, 0, THREAD\_SIZE);  
  
 auto mergeSize = getSplits(&splits, size, thr);  
  
 pthread\_t threads[splits.size()];  
  
 auto threadCount = 0;  
 int semVal;  
  
 while (!splits.empty()){  
 sem\_getvalue(&sem, &semVal);  
 auto p = splits.front();  
 splits.pop();  
 auto params = new InsertParams{values, p.first, p.second};  
 pthread\_create(&threads[threadCount], nullptr, insertSortThread, params);  
 ++threadCount;  
 }  
  
 for (auto j = 0; j < threadCount; j++){  
 pthread\_join(threads[j], nullptr);  
 }  
  
 while (mergeSize <= size){  
 threadCount = 0;  
 for (size\_t start = 0; start < size; start += 2 \* mergeSize) {  
 size\_t mid = std::min(start + mergeSize, size);  
 size\_t end = std::min(start + 2 \* mergeSize, size);  
  
 if (mid < end) {  
 auto params = new MergeParams{ values, start, mid, mid, end };  
 pthread\_create(&threads[threadCount], nullptr, mergeSortThread, params);  
 ++threadCount;  
 }  
 }  
  
 for (auto j = 0; j < threadCount; j++){  
 pthread\_join(threads[j], nullptr);  
 }  
  
 mergeSize \*= 2;  
 }  
  
 sem\_destroy(&sem);  
 }  
}

**Timsort.h**

#pragma once  
  
#include <cstdio>  
namespace timsort {  
 void sort(long \*values, size\_t size);  
 void sortAsync(long \*values, size\_t size, size\_t threads);  
}

**Test.cpp**

#include <gtest/gtest.h>  
#include <random>  
#include "../src/timsort.h"  
  
void generateArray(long\* values, size\_t size) {  
 std::random\_device rd;  
 std::mt19937 mt(rd());  
 std::uniform\_int\_distribution<long> dist(1, 20);  
  
 for (auto i = 0; i < size; ++i){  
 values[i] = dist(mt);  
 }  
}  
  
bool validateSort(const long\* values, size\_t size){  
 for (auto i = 0; i < size - 1; ++i){  
 if (values[i] > values[i + 1]) return false;  
 }  
  
 return true;  
}  
  
bool arrayEq(const long\* v1, const long\* v2, size\_t size){  
 for (auto i = 0; i < size; ++i){  
 if (v1[i] != v2[i]) return false;  
 }  
  
 return true;  
}  
  
void copyArray(const long\* in, long\* dest, size\_t size){  
 for (auto i = 0; i < size; ++i){  
 dest[i] = in[i];  
 }  
}  
  
TEST(SyncTests, Should\_SortSyncronouslyCorrectly){  
 const auto size = 1000;  
 auto values = static\_cast<long\*>(std::malloc(sizeof(long) \* size));  
 generateArray(values, size);  
 timsort::sort(values, size);  
 EXPECT\_TRUE(validateSort(values, size));  
 std::free(values);  
}  
  
TEST(SyncAsyncTests, Should\_ReturnSameValuesForSyncAndAsyncVersions){  
 const auto size = 1000;  
 auto values = static\_cast<long\*>(std::malloc(sizeof(long) \* size));  
 auto values1 = static\_cast<long\*>(std::malloc(sizeof(long) \* size));  
 generateArray(values, size);  
 copyArray(values, values1, size);  
  
  
 timsort::sort(values, size);  
 timsort::sortAsync(values1, size, 8);  
  
 EXPECT\_TRUE(arrayEq(values, values1, size));  
  
 std::free(values);  
 std::free(values1);  
}  
  
TEST(SyncAsyncTests, Should\_SortSyncronouslyFaster){  
 const auto size = 100000;  
 auto values = static\_cast<long\*>(std::malloc(sizeof(long) \* size));  
 auto values1 = static\_cast<long\*>(std::malloc(sizeof(long) \* size));  
 generateArray(values, size);  
 copyArray(values, values1, size);  
  
 const auto startSync{std::chrono::high\_resolution\_clock::now()};  
 timsort::sort(values, size);  
 const auto endSync{std::chrono::high\_resolution\_clock::now()};  
  
 auto sync = std::chrono::duration\_cast<std::chrono::microseconds>(endSync - startSync);  
  
 const auto startAsync{std::chrono::high\_resolution\_clock::now()};  
 timsort::sortAsync(values1, size, 8);  
 const auto endAsync{std::chrono::high\_resolution\_clock::now()};  
  
 auto async = std::chrono::duration\_cast<std::chrono::microseconds>(endAsync - startAsync);  
  
 EXPECT\_TRUE(arrayEq(values, values1, size));  
 EXPECT\_TRUE(async < sync);  
  
 std::free(values);  
 std::free(values1);  
}  
  
int main(int argc, char \*\*argv) {  
 testing::InitGoogleTest(&argc, argv);  
 return RUN\_ALL\_TESTS();  
}

**Main.cpp**

*//  
// Created by nikit on 10/25/2024.  
//*#include <cstdlib>  
#include <random>  
#include <iostream>  
#include "src/timsort.h"  
  
int main(int argc, char\*\* argv){  
 std::random\_device rd;  
 std::mt19937 mt(rd());  
 std::uniform\_int\_distribution<long> dist(1, 20);  
  
 size\_t size = 101;  
 auto values = static\_cast<long\*>(std::malloc(sizeof(long) \* size));  
 for (auto i = 0; i < size; ++i){  
 values[i] = dist(mt);  
 }  
  
 for (auto i = 0; i < size; ++i){  
 std::cout << values[i] << " ";  
 }  
  
 std::cout << std::endl;  
  
 timsort::sortAsync(values, size, std::atol(argv[1]));  
  
 for (auto i = 0; i < size; ++i){  
 std::cout << values[i] << " ";  
 }  
  
 std::cout << std::endl;  
  
 std::free(values);  
 return 0;  
}

**Заключение**

Выполнив лабораторную работу, я приобрел навыки управления и синхронизации потоков. Многопоточность существенно ускоряет сортировку массива методом TimSort, особенно при увеличении количества потоков и размера входных данных. Однако эффективность снижается с ростом числа потоков из-за накладных расходов на управление ими. Для оптимальной производительности рекомендуется использовать количество потоков, соответствующее числу ядер процессора, и применять многопоточность преимущественно к большим объемам данных, где преимущества параллельной обработки наиболее заметны.