Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Операционные среды и системное программирование»

|  |  |
| --- | --- |
|  |  |

**ОТЧЕТ**

к лабораторной работе № 5

на тему

**УПРАВЛЕНИЕ ПОТОКАМИ, СРЕДСТВА СИНХРОНИЗАЦИИ**

Выполнил             В. Д. Ключинский

Проверил                          Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc163819256)

[2.1 Понятие потока 4](#_Toc163819257)

[2.2 Преимущества и недостатки использования потоков 4](#_Toc163819258)

[2.3 Программирование потоков 6](#_Toc163819259)

[3 Полученные результаты 8](#_Toc163819260)

[Список использованных источников 11](#_Toc163819261)

[Приложение А](#_Toc163819262) [(обязательное)](#_Toc163819263) [Листинг исходного кода 12](#_Toc163819264)

**1 ЦЕЛЬ РАБОТЫ**

Целью выполнения лабораторной работы является изучение изучение подсистемы потоков (*pthread*), основных особенностей функционирования и управления, средств взаимодействия потоков. Практическое проектирование, реализация и отладка программ с параллельными взаимодействующими (конкурирующими) потоками на примере реализации многопоточной сортировки merge sort.

**2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

## **2.1. Понятие потока**

Поток (thread) можно определить как часть процесса, включающая управляющую последовательность команд и использующая системные ресурсы этого процесса. Существует две основных категории реализации потоков: Пользовательские потоки – потоки, реализуемые через специальные библиотеки потоков и работающие в пользовательском пространстве. Потоки уровня ядра – потоки, реализуемые через системные вызовы и работающие в пространстве ядра. Каждый процесс имеет, как минимум, один поток, при этом самый первый поток, создаваемый при рождении нового процесса, принято называть начальным или главным потоком этого процесса. Основное отличие процесса от потока заключается в способе использования системных ресурсов. Дочерний процесс практически независим от родительского, для него системой выделяется отдельное адресное пространство, и он на равных правах с родительским процессом «конкурирует» за процессорное время. При этом можно уничтожить родительский процесс, не затронув дочерний, который может выполняться и после завершения родительского процесса. В отличие от дочернего процесса, поток, порожденный данным процессом, полностью зависим от процесса и завершение процесса влечет уничтожение всех созданных им потоков, поскольку происходит освобождение системных ресурсов, выделенных для этого процесса. Еще одним отличием является невозможность изменить права доступа для потока, тогда как дочерний процесс в редких случаях (например, при смене пароля для входа в систему) может обладать правами доступа отличными от прав родительского процесса. Первая подсистема потоков в Linux появилась в 1996 году и называлась LinuxThreads, ее автором является Ксавье Лерой (Xavier Leroy). Разработанная им библиотека LinuxThreads была попыткой организовать поддержку потоков в Linux в то время, когда ядро системы еще не предоставляло никаких специальных механизмов для работы с потоками. Позднее разработку потоков для Linux вели сразу две конкурирующие группы – NGPT (New Generation POSIX Threads) и NPTL (Native POSIX Thread Library). В 2002 году группа NGPT фактически присоединилась к NPTL и теперь реализация потоков NPTL является стандартом Linux. Подсистема потоков Linux стремится соответствовать требованиям стандартов POSIX, поэтому новые многопоточные приложения Linux должны без проблем компилироваться на новых POSIX-совместимых системах.

## **2.2 Преимущества и недостатки использования потоков**

Потоки часто становятся источниками программных ошибок особого рода. Эти ошибки возникают при использовании потоками разделяемых ресурсов системы (например, общего адресного пространства) и являются частным случаем более широкого класса ошибок – ошибок синхронизации. Если задача разделена между независимыми процессами, то доступом к их общим ресурсам управляет ОС, и вероятность ошибок из-за конфликтов доступа снижается. Впрочем, разделение задачи между несколькими независимыми процессами само по себе не защищает от других разновидностей ошибок синхронизации. В пользу потоков можно указать то, что накладные расходы на создание нового потока в многопоточном приложении ниже, чем накладные расходы на создание нового самостоятельного процесса. Уровень контроля со стороны родительского процесса над порожденными потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, в результате работы многопоточных программ, как правило, не остаются незавершенными дочерние процессы, 57 зависящие от родительского (процессы-«зомби»), как может произойти при использовании многопроцессного приложения. Преимущества и недостатки использования нескольких потоков во время выполнения программы приведены в табл. 2.1.

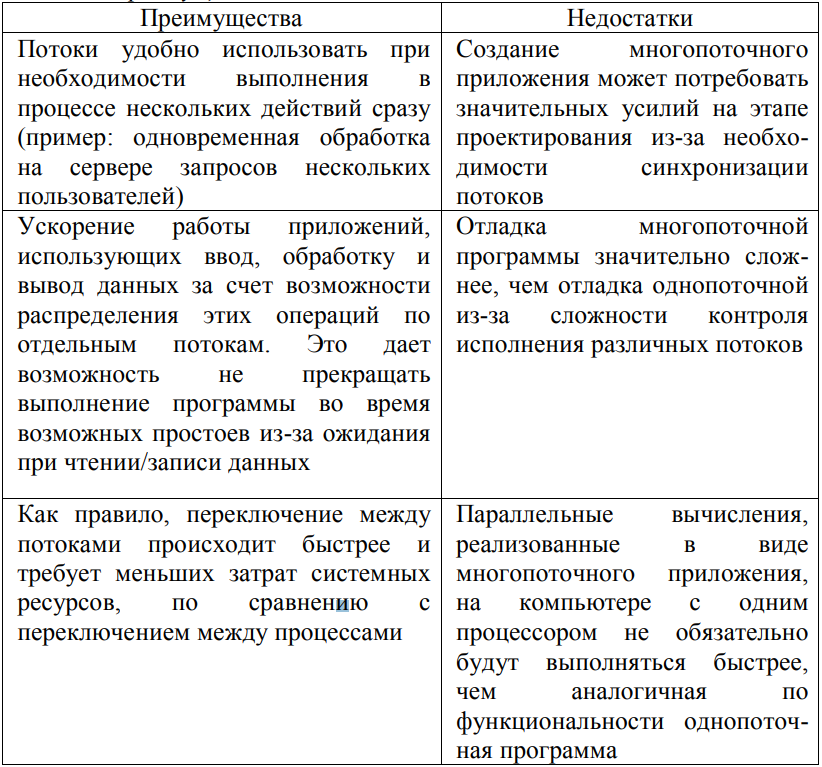


Таблица 2.1 – Преимущества и недостатки использования потоков

## **2.3 Программирование потоков**

В Linux каждый поток на самом деле является процессом, и для того, чтобы создать новый поток, нужно создать новый процесс. Однако для создания дополнительных потоков используются процессы особого типа. Эти процессы представляют собой обычные дочерние процессы главного процесса, но они разделяют с главным процессом адресное пространство, файловые дескрипторы и обработчики сигналов. Для обозначения процессов этого типа применяется специальный термин – легкие процессы (*lightweight processes*).

Поскольку для потоков не требуется создавать собственную копию адресного пространства (и других ресурсов) своего процесса-родителя, создание нового легкого процесса требует значительно меньших затрат, чем создание полновесного дочернего процесса. Спецификация POSIX 1003.1c требует, чтобы все потоки многопоточного приложения имели один идентификатор, однако в Linux у каждого процесса, в том числе и у процессов-потоков, есть свой идентификатор.

**Основные функции для работы с потоками**

Для работы с потоками используются следующие основные функции:

1 pthread\_create – создание потока;

2 pthread\_join – блокирование работы вызвавшего функцию процесса или потока в ожидании завершения потока;

3 pthread\_cancel – досрочное завершение потока из другого потока или процесса;

4 pthread\_exit – завершает поток, код завершения передается функции pthread\_join. Данная функция подобна функции exit, однако вызов exit в «основном» процессе программы приведет к завершению всей программы.

**Запуск и завершение потока**

Потоки создаются функцией pthread\_create, имеющей следующую сигнатуру int pthread\_create (pthread\_t\* tid, pthread\_attr\_t\* attr, void\*(\*function)(void\*), void\* arg). Данная функция определена в заголовочном файле . Первый параметр этой функции представляет собой указатель на переменную типа pthread\_t, которая служит идентификатором создаваемого потока. Второй параметр – указатель на переменную типа pthread\_attr\_t – используется для установки атрибутов потока. Третьим параметром функции pthread\_create должен быть адрес функции потока. Эта функция играет для потока ту же роль, что функция main для главной программы. Четвертый параметр функции pthread\_create имеет тип void\*. Этот параметр может использоваться для передачи значения в функцию потока. Вскоре после вызова pthread\_create функция потока будет запущена на выполнение параллельно с другими потоками программы. Новый поток запускается не сразу после вызова pthread\_create, потому что перед тем, как запустить новую функцию потока, нужно выполнить некоторые подготовительные действия, а поток-родитель при этом продолжает выполняться. Необходимо учитывать данное обстоятельство при разработке многопоточного приложения, в противном случае возможны серьезные ошибки при выполнении программы. Если при создании потока возникла ошибка, то функция pthread\_create возвращает ненулевое значение, соответствующее номеру ошибки. Функция потока должна иметь сигнатуру вида void\* func\_name(void\* arg). Имя функции может быть любым. Аргумент arg является указателем, который передается в последнем параметре функции pthread\_create. Функция потока может вернуть значение, которое затем может быть обработано другим потоком или процессом. Функция, вызываемая из функции потока, должна обладать свойством реентерабельности (этим же свойством должны обладать рекурсивные функции). Реентерабельная функция – это функция, которая может быть вызвана повторно, в то время, когда она уже вызвана. Такие функции используют локальные переменные (и локально выделенную память) в тех случаях, когда их нереентерабельные аналоги могут воспользоваться глобальными переменными.

Завершение функции потока происходит в следующих случаях:

– функция потока вызвала функцию pthread\_exit;

– функция потока достигла точки выхода;

– поток был досрочно завершен другим потоком или процессом.

# **3 ПОЛУЧЕННЫЕ РЕЗУЛЬТАТЫ**

В результате выполнения лабораторной работы была создана программа, выполняющая сортировку целых чисел методом merge sort в многопоточном режиме. Всё начальное множество разбивается на равные интервалы, где каждый интервал обрабатывается в отдельном потоке. Для каждой пары отсортированных интервалов создается новый поток, который производит их объединение. Так продолжается до тех пор, пока все интервалы не объединятся в один. Данная программа была создана средствами языка C++. Скомпилирована с помощью makefile.

Результаты работы программы и исходные файлы представлены на рисунках ниже.

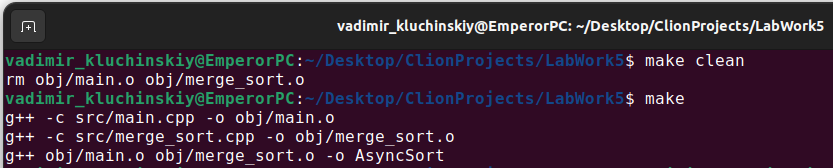


Рисунок 3.1 – Процесс компиляции программы с помощью makefile

В следующем примере была выполнена сортировка однопоточным и многопоточным способом, который сортируют 10000 случайных чисел используя 16 потоков. Как видно из рисунка 3.2 многопоточный режим даёт преимущество в 2.95 раза, что является серьёзным результатом.

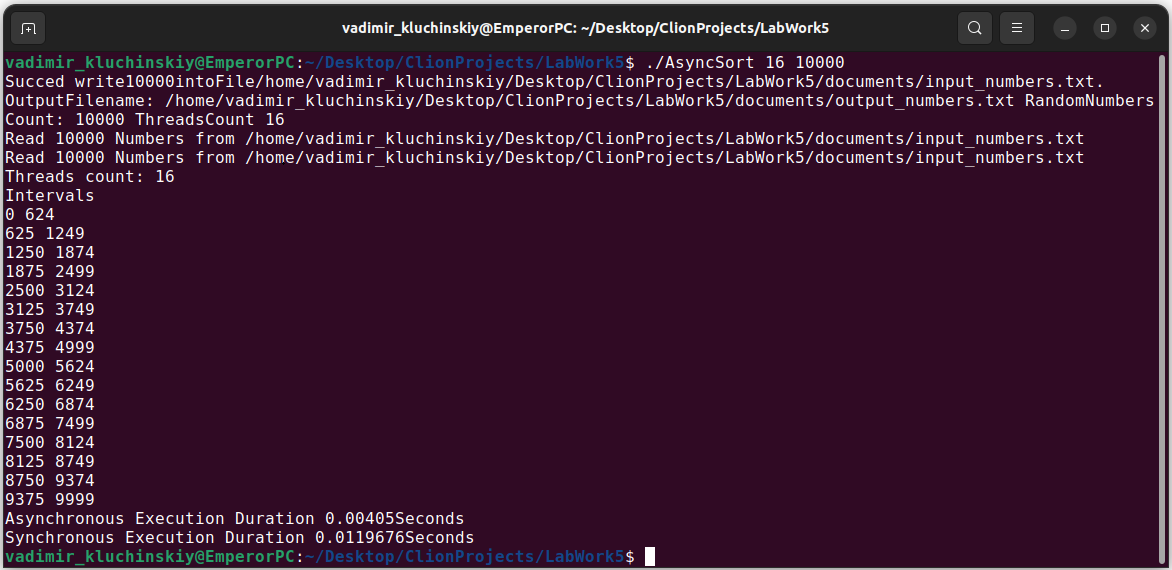


Рисунок 3.2 – Результат сравнения при 10000 числах

Также было произведено сравнение при количестве чисел равном 4 миллиона. Разница в скорости в данном случае составила 4.862, что можно видеть на рисунке 3.3

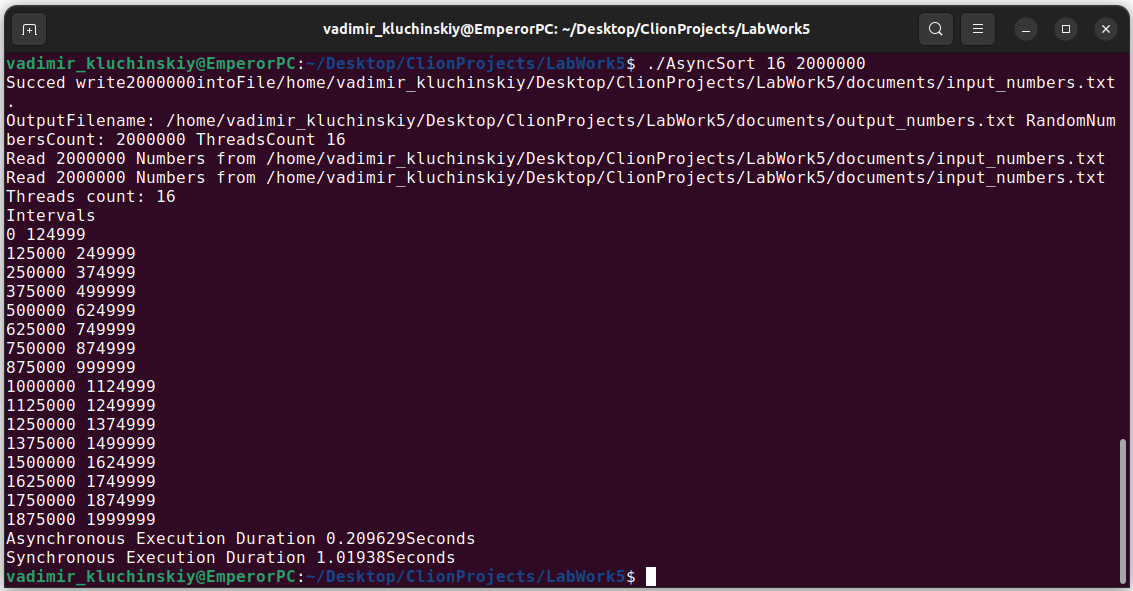


Рисунок 3.3 – Результат сравнения при 2000000

Отсортированный массив записывается в текстовый файл. Что можно видеть на рисунке ниже.

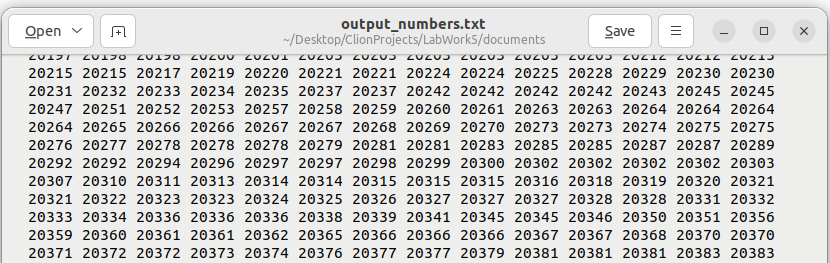


Рисунок 3.4 – Текстовый файл, содержащий отсортированный массив

Таким образом, созданная программа полностью выполняет поставленную задачу

**ВЫВОДЫ**

В ходе выполнения данной лабораторный работы была создана программа, которая выполняет многопоточную сортировку методом merge sort для большого набора целых чисел на языке C++. Были изучены механизмы создания и синхронизации потоков в операционной системе Linux.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Тейнсли, Д. Linux и другие UNIX-подобные операционные системы: программирование. / Д. Тейнсли. – СПб.: БВХ-Петербург, 2006. – 1056 с.

[2] Щупаков, Ю. Руководство по командам и shell-программированию, операционная система Linux. / Ю. Щупаков. – СПб: Питер, 2008. – 592 с.

[3] UnixTutorials [Электронный ресурс]. – Режим доступа: https://prog /unix-tutorials/. – Дата доступа: 06.02.2024.

# **ПРИЛОЖЕНИЕ А**

# **(обязательное)**

# **Листинг исходного кода**

Листинг 1 – Программный код разработанного приложения:

Файл *merge\_sort.h*:

#ifndef LABWORK5\_MERGE\_SORT\_H

#define LABWORK5\_MERGE\_SORT\_H

#include <iostream>

#include <thread>

#include<chrono>

#include<vector>

#include<deque>

#include<fstream>

#include <random>

using namespace std;

struct merge\_structure{

std::vector<int>& arr;

int left, middle, right;

};

void display\_vec(vector<int>& vec);

int write\_random\_nums\_to\_file(const char\* filename, size\_t num);

void read\_numbers(vector<int>& numbers, const char\* filename);

int write\_numbers(vector<int>& numbers, const char\* filename);

void merge2(merge\_structure&& ms);

void merge\_sort(std::vector<int>& arr, int left, int right);

void async\_merge\_sort42(vector<int>& array, size\_t num\_intervals);

#endif //LABWORK5\_MERGE\_SORT\_H

Файл *merge\_sort.cpp*:

#ifndef LABWORK5\_MERGE\_SORT\_CPP

#define LABWORK5\_MERGE\_SORT\_CPP

#include "../headers/merge\_sort.h"

void display\_vec(vector<int>& vec) {

for (auto& now : vec) {

cout << now << ' ';

}

cout << '\n';

}

int write\_random\_nums\_to\_file(const char\* filename, size\_t numbers\_count ){

std::fstream outputFile(filename, ios::trunc|ios::binary|ios::out);

if (!outputFile.is\_open()) {

std::cerr << "error writing to file" <<'\n';

return 1;

}

vector<int> vec(numbers\_count);

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(1, numbers\_count); // Çäåñü 1 è 1000 - äèàïàçîí ñëó÷àéíûõ ÷èñåë

vec.resize(numbers\_count);

for (int i = 0; i < numbers\_count; ++i) {

vec[i] = dis(gen);

}

for (auto& num: vec) {

outputFile << num << ' ';

}

outputFile.close();

std::cout << "Succed write" << numbers\_count << "intoFile" << filename << "." << std::endl;

return 0;

}

void read\_numbers(vector<int>& numbers, const char\* filename) {

std::ifstream inputFile(filename);

if (!inputFile.is\_open()) {

std::cerr << "Failed to read" <<'\n';

return;

}

int number;

while (inputFile >> number) {

numbers.push\_back(number);

}

inputFile.close();

std::cout << "Read " << numbers.size() << " Numbers from " << filename <<'\n';

}

int write\_numbers(vector<int>& numbers, const char\* filename){

std::fstream outputFile(filename, ios::trunc|ios::binary|ios::out);

if (!outputFile.is\_open()) {

std::cerr << "error writing to file" <<'\n';

return 1;

}

for (auto& num: numbers) {

outputFile << num << ' ';

}

outputFile.close();

return 0;

}

void merge2(merge\_structure&& ms) {

vector<int>& arr = ms.arr;

int left = ms.left, middle = ms.middle, right = ms.right;

int n1 = middle - left + 1, n2 = right - middle;

vector<int> leftArr(n1);

vector<int> rightArr(n2);

for (int i = 0; i < n1; ++i) {

leftArr[i] = arr[left + i];

}

for (int j = 0; j < n2; ++j) {

rightArr[j] = arr[middle + 1 + j];

}

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) {

arr[k] = leftArr[i];

++i;

}

else {

arr[k] = rightArr[j];

++j;

}

++k;

}

while (i < n1) {

arr[k] = leftArr[i];

++i;

++k;

}

while (j < n2) {

arr[k] = rightArr[j];

++j;

++k;

}

}

void merge\_sort(std::vector<int>& arr, int left, int right) {

if (left < right) {

int middle = left + (right - left) / 2;

merge\_sort(arr, left, middle);

merge\_sort(arr, middle + 1, right);

merge\_structure ms{ref(arr), left, middle, right};

merge2(merge\_structure{ref(arr), left, middle, right});

//merge2(ms);

}

}

void async\_merge\_sort42(vector<int>& array, size\_t num\_intervals) {

cout << "Threads count: " << num\_intervals << '\n';

//std::vector<int> array = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 }; // Âàø ìàññèâ

size\_t size = array.size(); // Êîëè÷åñòâî èíòåðâàëîâ

int step = size / num\_intervals - 1; // Ðàçìåð êàæäîãî èíòåðâàëà

vector<pair<int, int>> intervals(num\_intervals);

intervals[0].first = 0, intervals[0].second = step;

for (int i = 1; i < num\_intervals; ++i) {

intervals[i].first = intervals[i - 1].second + 1;

intervals[i].second = intervals[i].first + step;

}

intervals[num\_intervals - 1].second = size - 1;

cout << "Intervals\n";

for (auto& now : intervals) {

cout << now.first << ' ' << now.second << '\n';

}

vector<thread> threads(num\_intervals);

for (int i = 0; i < num\_intervals; ++i) {

threads[i] = thread(merge\_sort, ref(array), intervals[i].first, intervals[i].second);

}

/\*for (int i = 0; i < num\_intervals; ++i) {

threads[i].join();

}\*/

while (intervals.size() >= 2) {

int count\_intervals = intervals.size();

int count\_threads = threads.size();

vector<pair<int, int>> second\_intervals;

vector<thread> second\_threads;

for (int i = 0; i < count\_intervals - 1; i += 2) {//2 transform to one

threads[i].join();

if (count\_threads >= i + 2) {

threads[i + 1].join();

}

merge\_structure ms{ref(array), intervals[i].first, intervals[i].second, intervals[i + 1].second};

second\_threads.push\_back(thread(merge2,ms));

second\_intervals.push\_back({ intervals[i].first, intervals[i+1].second });

}

if (count\_intervals & 1) {

int last = count\_intervals - 1;

if (count\_threads >= last+1) {

threads[last].join();

}

second\_intervals.push\_back(intervals[last]);

}

threads.clear();

threads = std::move(second\_threads);

intervals = second\_intervals;

}

threads[0].join();

/\*for (auto& now : second\_threads) {

now.join();

}\*/

/\* merge(array, intervals[0].first, intervals[1].second, intervals[3].second);\*/

}

#endif //LABWORK5\_MERGE\_SORT\_CPP

Файл *main.cpp*:

#include "../headers/merge\_sort.h"

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

{

const char\* input\_filename = "/home/vadimir\_kluchinskiy/Desktop/ClionProjects/LabWork5/documents/input\_numbers.txt";

const char\* output\_filename = "/home/vadimir\_kluchinskiy/Desktop/ClionProjects/LabWork5/documents/output\_numbers.txt";

vector<int> vec,vec1;

size\_t nums\_count=10000, threds\_number = 8;

if(argc == 2){

threds\_number = stoul(argv[1]);

}

else if (argc == 3) {

threds\_number = stoul(argv[1]);

nums\_count = stoul(argv[2]);

write\_random\_nums\_to\_file(input\_filename,nums\_count);

}

printf("OutputFilename: %s RandomNumbersCount: %zu ThreadsCount %zu\n", output\_filename, nums\_count, threds\_number);

read\_numbers(vec,input\_filename);

read\_numbers(vec1,input\_filename);

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

async\_merge\_sort42(vec, threds\_number);

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

// display\_vec(vec);

auto start1 = chrono::high\_resolution\_clock::now();

merge\_sort(vec1, 0, vec1.size() - 1);

auto end1 = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration1 = end1 - start1;

cout << "Asynchronous Execution Duration " << duration.count() << "Seconds\n";

cout << "Synchronous Execution Duration " << duration1.count() << "Seconds\n";

write\_numbers(vec, output\_filename);

return 0;