Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе № 5

по курсу «Анализ алгоритмов»

на тему: «Организация асинхронного взаимодействия потоков вычисления на примере конвейерных вычислений»

Студент ИУ7-54Б (Группа) _____ (Подпись, дата) Разин А. (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) Волкова Л. Л. (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Разработчики архитектуры компьютеров издавна прибегали к методам проектирования, известным под общим названием «совмещение операций», при котором аппаратура компьютера в любой момент времени выполняет одновременно более одной базовой операции [1].

Этот метод включает в себя, в частности, такое понятие, как конвейеризация. Конвейеры широко применяются программистами для решения трудоемких задач, которые можно разделить на этапы, а также в большинстве современных быстродействующих процессоров [1].

В качестве операций, выполняющихся на конвейере в данной работе, взяты следующие:

1. сортировка слиянием копии массива, предоставленного в заявке;
2. сортировка слиянием исходного массива с использованием 4 дополнительных потоков;
3. запись значений отсортированного массива в файл, соответствующий номеру заявки.

Целью данной работы является получение навыков организации конвейерной обработки данных.

В рамках выполнения работы необходимо решить следующие задачи:

1. описать организацию конвейерной обработки данных;
2. описать алгоритмы обработки данных, которые будут использоваться в текущей лабораторной работе;
3. реализовать программу, выполняющую конвейерную обработку с количеством лент не менее трех и программу обрабатывающую заявки последовательно;
4. провести сравнительный анализ времени работы реализаций.

1 Аналитическая часть

В данной части работы будут рассмотрены основы конвейерной обработки данных, а также будет описан алгоритм сортировки слиянием.

1.1 Конвейерная обработка данных

Конвейеризация (или конвейерная обработка) в общем случае основана на разделении подлежащей исполнению функции на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры. Производительность при этом возрастает благодаря тому, что одновременно на различных ступенях конвейера выполняются несколько команд. Конвейерная обработка такого рода широко применяется во всех современных быстродействующих процессорах [2].

1.2 Сортировка слиянием

Алгоритм сортировки слиянием (англ. merge sort) является эффективным и стабильным алгоритмом сортировки, который применяет принцип «разделяй и властвуй» для упорядочивание элементов в массиве [3].

Алгоритм состоит из нескольких этапов [3].

1. Разделение — исходный массив разделяется на две равные (или почти равные) половины. Это делается путем нахождения середины массива и создания двух новых массивов, в которые будут скопированы элементы из левой и правой половин.

2. Рекурсивная сортировка: каждая половина массива рекурсивно сортируется с помощью алгоритма сортировки слиянием. Этот шаг повторяется до тех пор, пока размер каждой половины не станет равным 1.

3. Слияние: отсортированные половины массива объединяются в один отсортированный массив. Для этого создается новый массив, в который будут последовательно добавляться элементы из левой и правой половин. При добавлении элементов выбирается наименьший элемент из двух половин и добавляется в новый массив. Этот шаг повторяется до тех пор, пока все элементы не будут добавлены в новый массив.

В результате выполнения этих шагов получается отсортированный массив, который содержит все элементы исходного массива.

Вывод

В данной части работы была рассмотрена конвейерная обработка данных и алгоритм сортировки слиянием, а также описаны конкретные операции, выполняющиеся в данной работе при конвейерной обработке.

2 Конструкторская часть

В данной части работы будут рассмотрены схемы алгоритмов конвейерной и линейной обработки заявок по сортировке массивов.

2.1 Разработка алгоритмов

На рисунках 2.1–2.5 приведены схемы алгоритмов линейной обработки заявок и конвейерной обработки заявок, также представлена схема каждого этапа обработки заявок. При конвейерной обработке заявок, создается 3 потока, каждый выполняет один из этапов обработки и продолжает выполнение, пока не получит отмеченную заявку, являющуюся последней из всех рассматриваемых заявок.

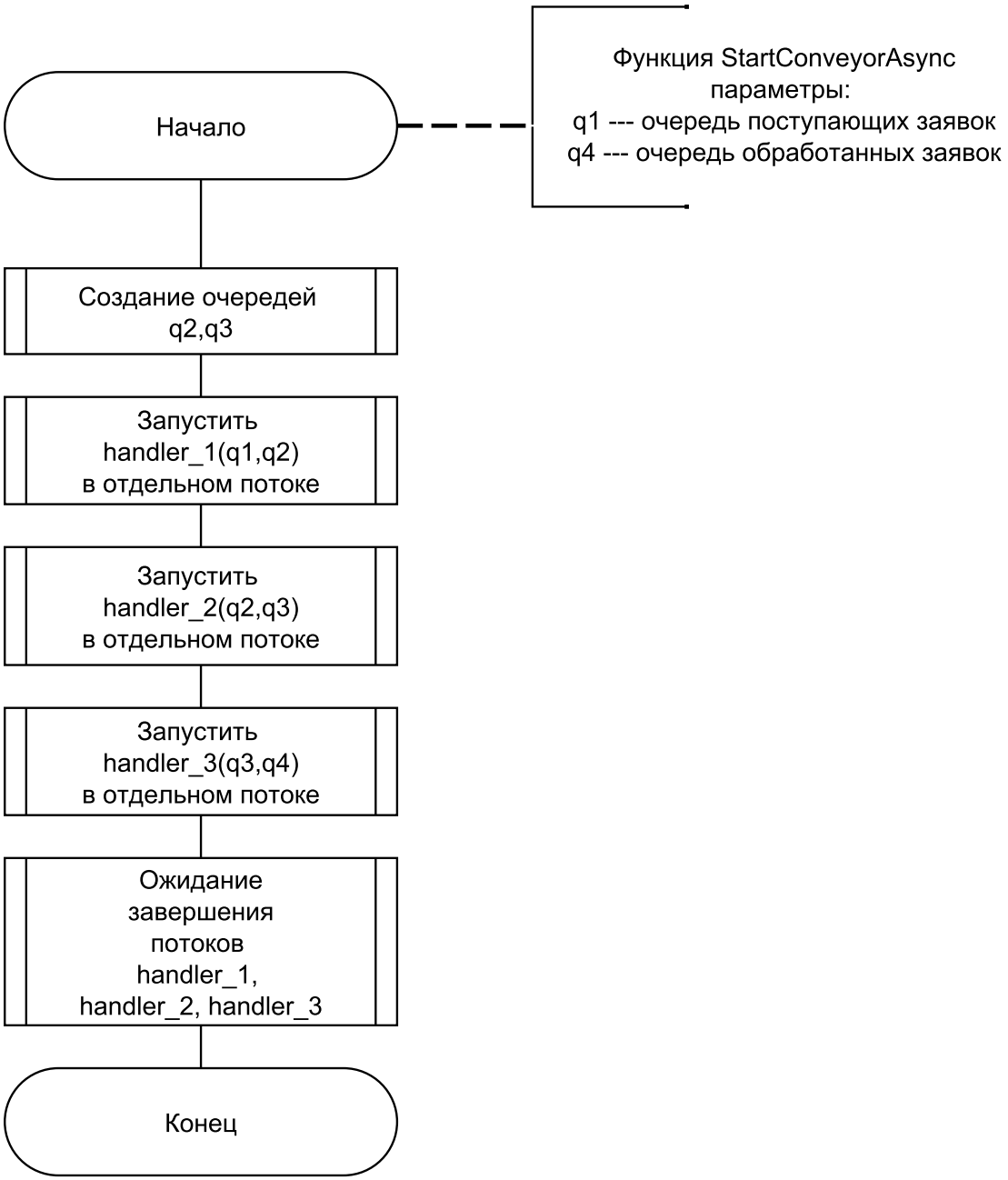

Рисунок 2.2 – Схема алгоритма конвейерной обработки заявок

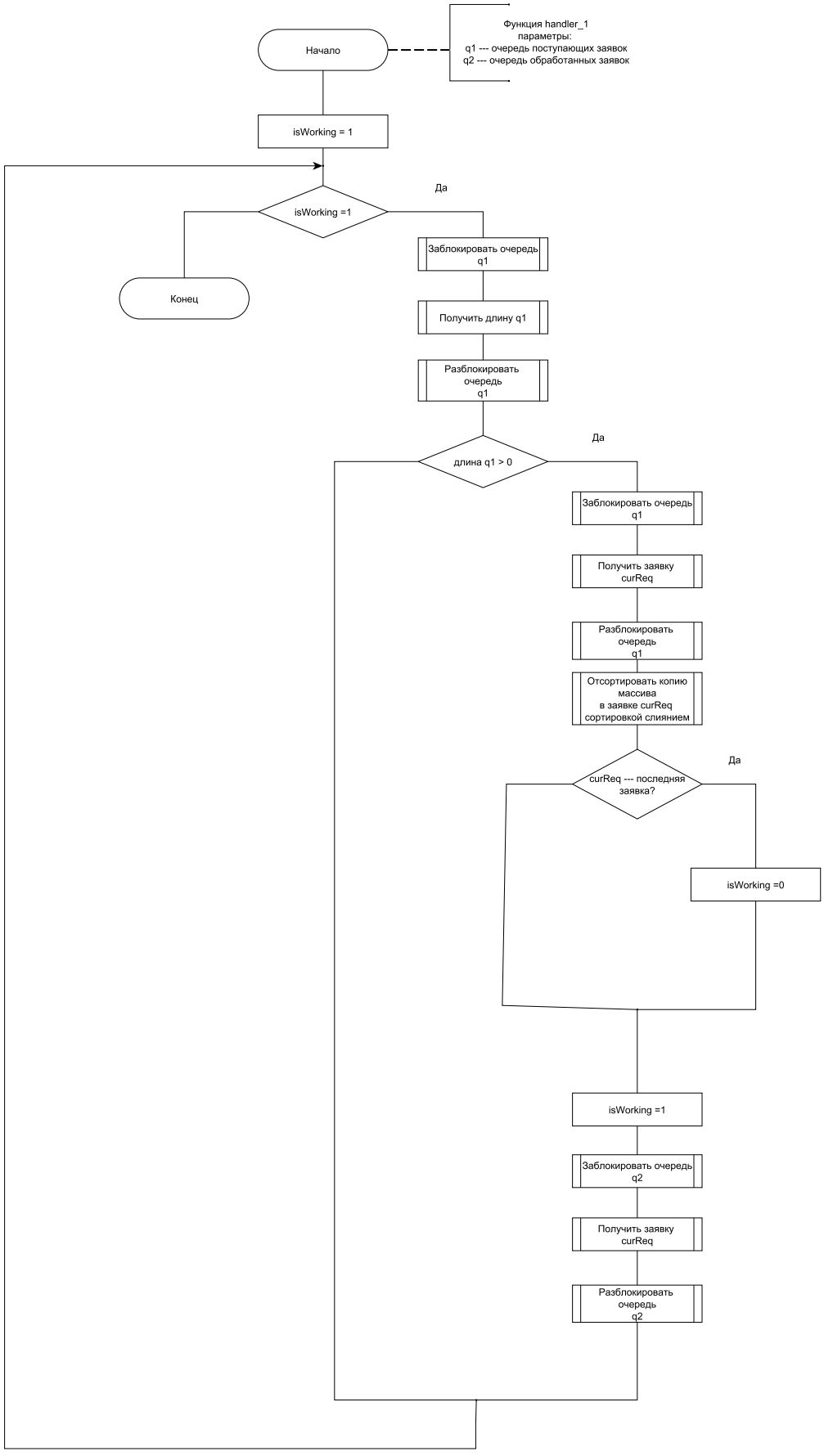

Рисунок 2.3 – Схема работы потока на 1 этапе конвейерных вычислений (сортировка копии массива)

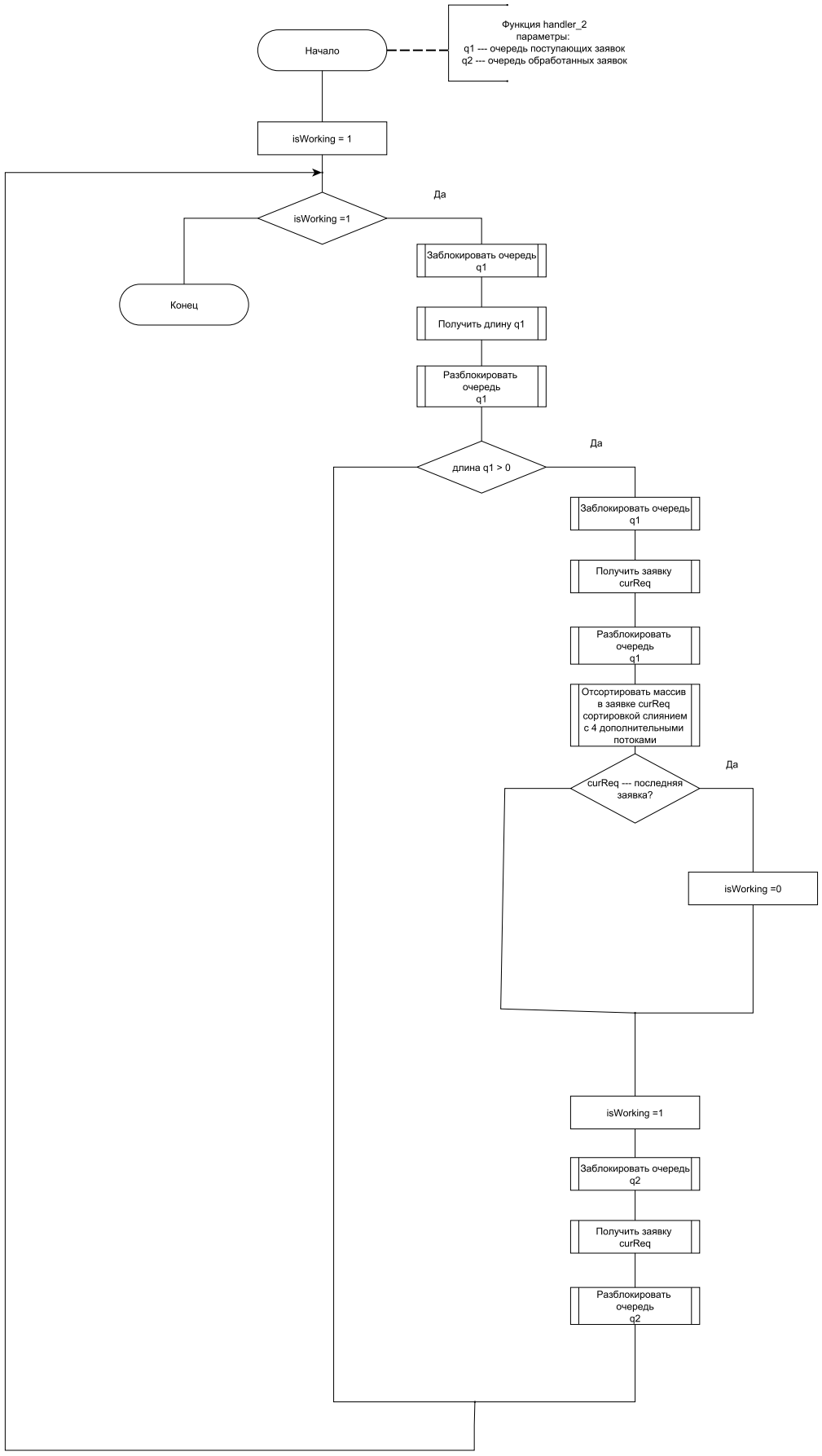

Рисунок 2.4 – Схема работы потока на 2 этапе конвейерных вычислений (сортировка массива с использованием дополнительных потоков)

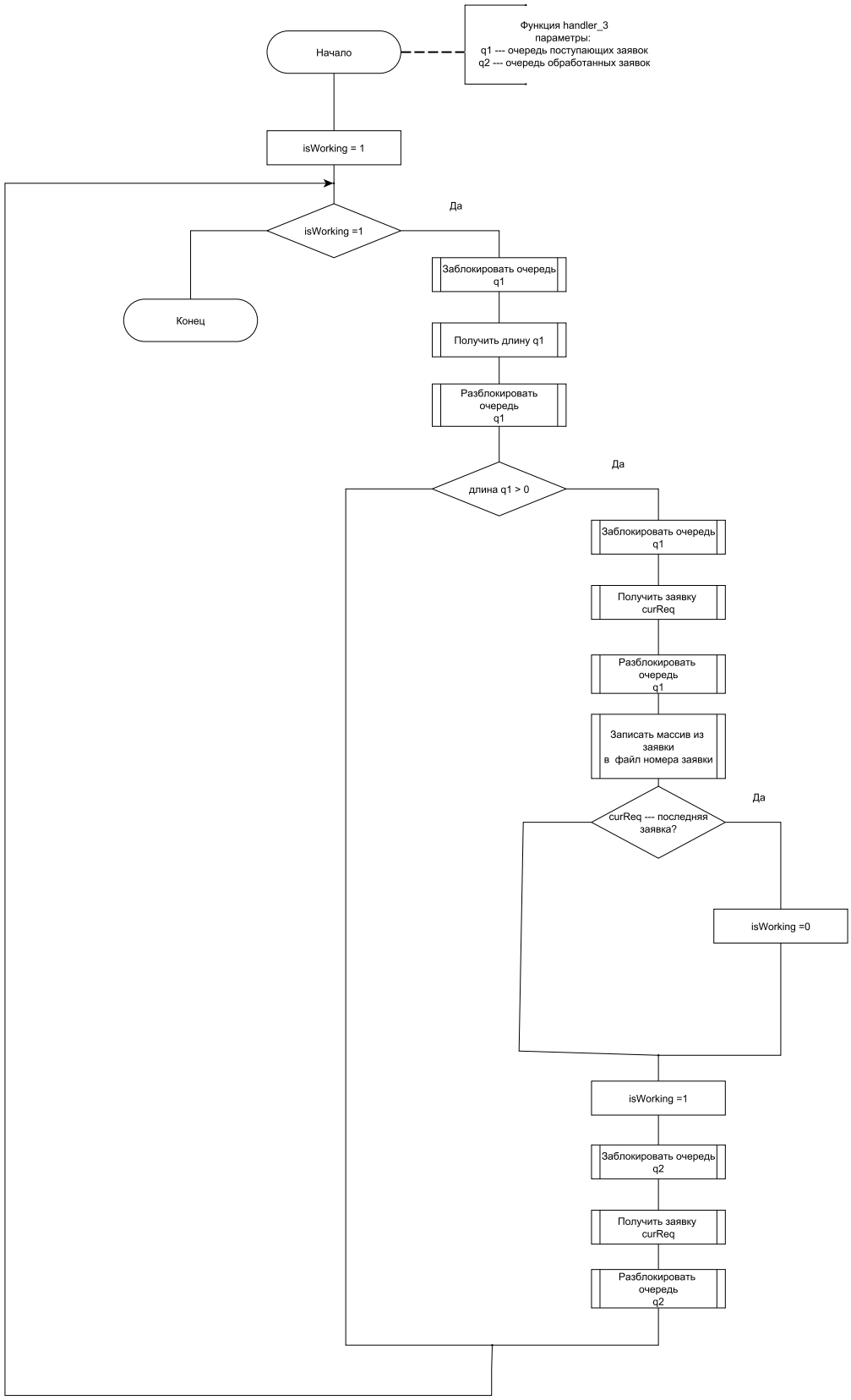

Рисунок 2.5 – Схема работы потока на 3 этапе конвейерных вычислений (запись массива в файл)

Вывод

В данной части работы были построены схемы алгоритмов последовательной и конвейерной обработки данных, а также отдельно рассмотрен каждый этап обработки заявки.

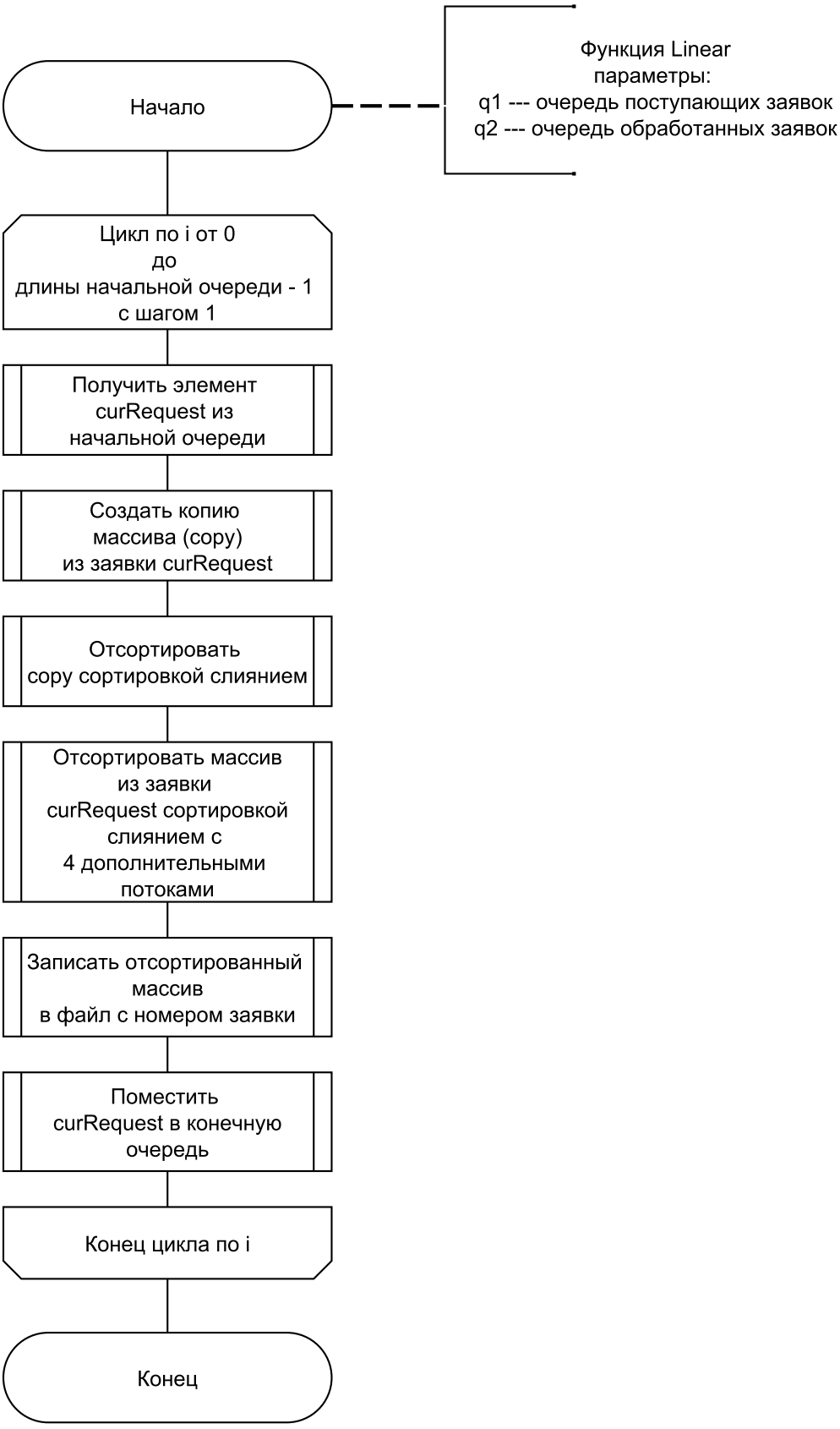

Рисунок 2.1 – Схема линейного алгоритма обработки заявок

3 Технологический раздел

В данной части работы будут описаны средства реализации программы, а также листинги и функциональные тесты.

3.1 Средства реализации

Алгоритмы для данной лабораторной работы были реализованы на языке C++, при использовании компилятора gcc версии 10.5.0, так как в стандартной библиотеке приведенного языка присутствует функция `clock_gettime`, которая позволяет получить реальное время а также класс `queue`, упрощающий использование очереди [4; 5].

3.2 Реализация алгоритмов

Листинги исходных кодов программ А.1–А.6 приведены в приложении.

3.3 Тестирование

В таблице 3.1 приведены функциональные тесты для разработанных алгоритмов, фактический результат был получен на обеих реализациях обработок заявок. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Число заявок	Размер Массива	Ожидаемый р-т	Фактический р-т
2	-2	Вывод предупреждения	Вывод предупреждения
-2	2	Вывод предупреждения	Вывод предупреждения
2	2	Вывод лога	Вывод лога
80	10	Вывод лога	Вывод лога

Вывод

В данной части работы были представлены листинги реализованных алгоритмов и тесты, успешно пройденные программой.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, проведены замеры и сделан сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени, представлены далее.

1. Процессор Intel(R) Core(TM) i7-9750H CPU 2592 МГц, ядер: 6, логических процессоров: 12.
2. Оперативная память: 16 ГБ.
3. Операционная система: Майкрософт Windows 10 Pro [6].
4. Использованная подсистема: WSL2 [7].

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного приложения для алгоритмов сортировок.

```
Меню:
0)Выход
1)Запустить последовательную обработку массивов
2)Запустить конвейерную обработку массивов
3)Замеры времени реализаций

Введите пункт из меню: 3

Введите начальный размер массива, конечный размер и шаг изменения размера массива
10 25 10
Введите начальное число заявок, конечное число заявок и шаг изменения числа заявок
10 25 10
req | n | Linear | Async|
10|10|0.006433|0.0034736|
20|10|0.0111242|0.0093121|
10|20|0.0057218|0.0046108|
20|20|0.0115979|0.0103303|
Меню:
0)Выход
1)Запустить последовательную обработку массивов
2)Запустить конвейерную обработку массивов
3)Замеры времени реализаций
```

Рисунок 4.1 – Пример работы программы

4.3 Временные характеристики

Результаты замеров времени получения результатов от числа заявок, приведен в таблице 4.1, размер сортируемых массивов был равен 9000. В качестве сортируемых значений использовались числа от 1 до 10000000. Данные генерировались из равномерного распределения, время замеров приведено в секундах. По таблице 4.1, был получен график 4.2.

Таблица 4.1 – Замеры по времени получения результата реализаций различных способов обработки заявок от числа заявок

Заявки	Последовательная обработка (с)	Конвейерная обработка (с)
10	0.138498	0.132937
20	0.276232	0.270415
30	0.419785	0.375741
40	0.560126	0.53251
50	0.706062	0.620642
60	0.836043	0.734683
70	0.957982	0.866383
80	1.10419	0.988661
90	1.23092	1.11383
100	1.39349	1.27446
110	1.52715	1.41736
120	2.09141	1.93223

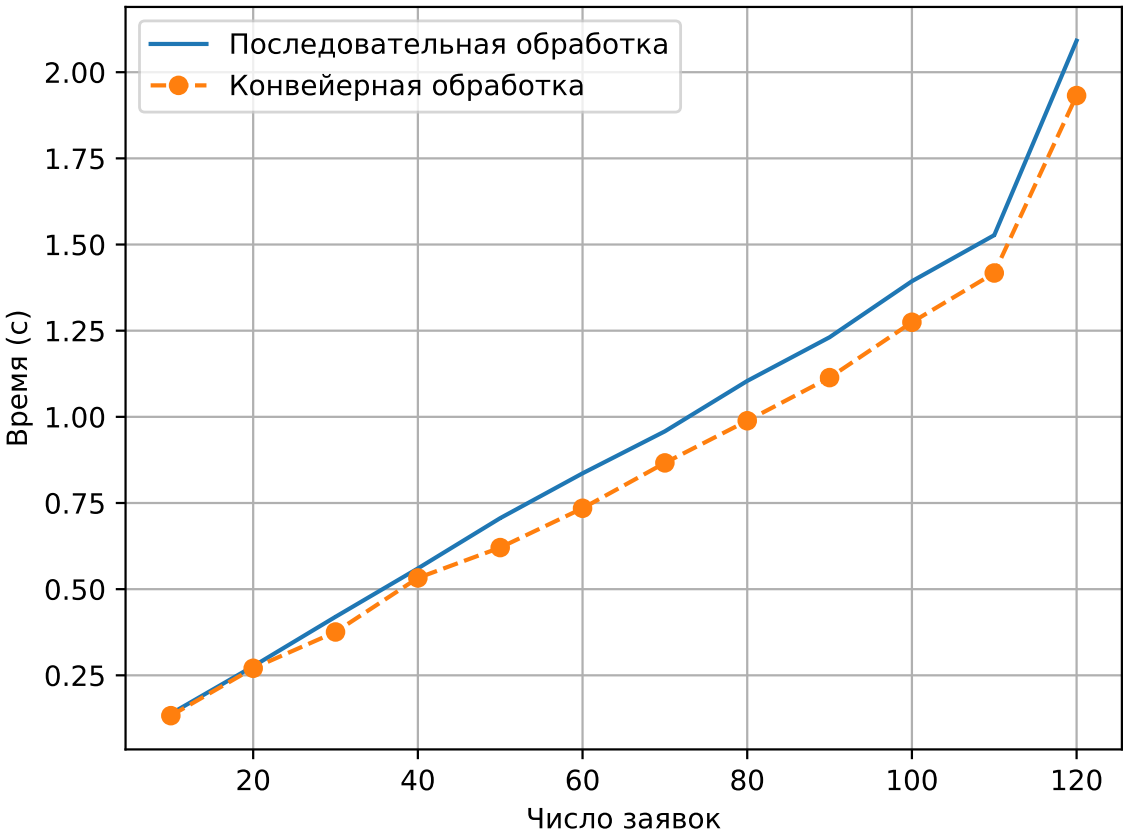

Рисунок 4.2 – Сравнение реализаций обработок заявок по времени получения результата в зависимости от числа заявок

Результаты замеров времени получения результатов от числа сортируемых элементов, приведен в таблице 4.2, число заявок было равно 100, столбец n в данном случае описывает число сортируемых элементов. В качестве сортируемых значений использовались числа от 1 до 10000000, время замеров приведено в секундах. Данные генерировались из равномерного распределения. По таблице 4.2, был получен график 4.3.

Таблица 4.2 – Замеры по времени получения результата реализаций различных способов обработки заявок от числа элементов в массиве

n	Последовательная обработка (с)	Конвейерная обработка (с)
1000	0.167468	0.108578
2000	0.291374	0.224196
3000	0.440222	0.362134
4000	0.557684	0.453164
5000	0.769051	0.63144
6000	0.884096	0.750595
7000	0.987792	0.851104
8000	1.16747	1.10913
9000	1.33448	1.30057
10000	1.62122	1.49942
11000	1.65712	1.57924
12000	1.80122	1.73396
13000	1.97171	1.85373
14000	2.08562	2.01101
15000	2.12513	1.96112

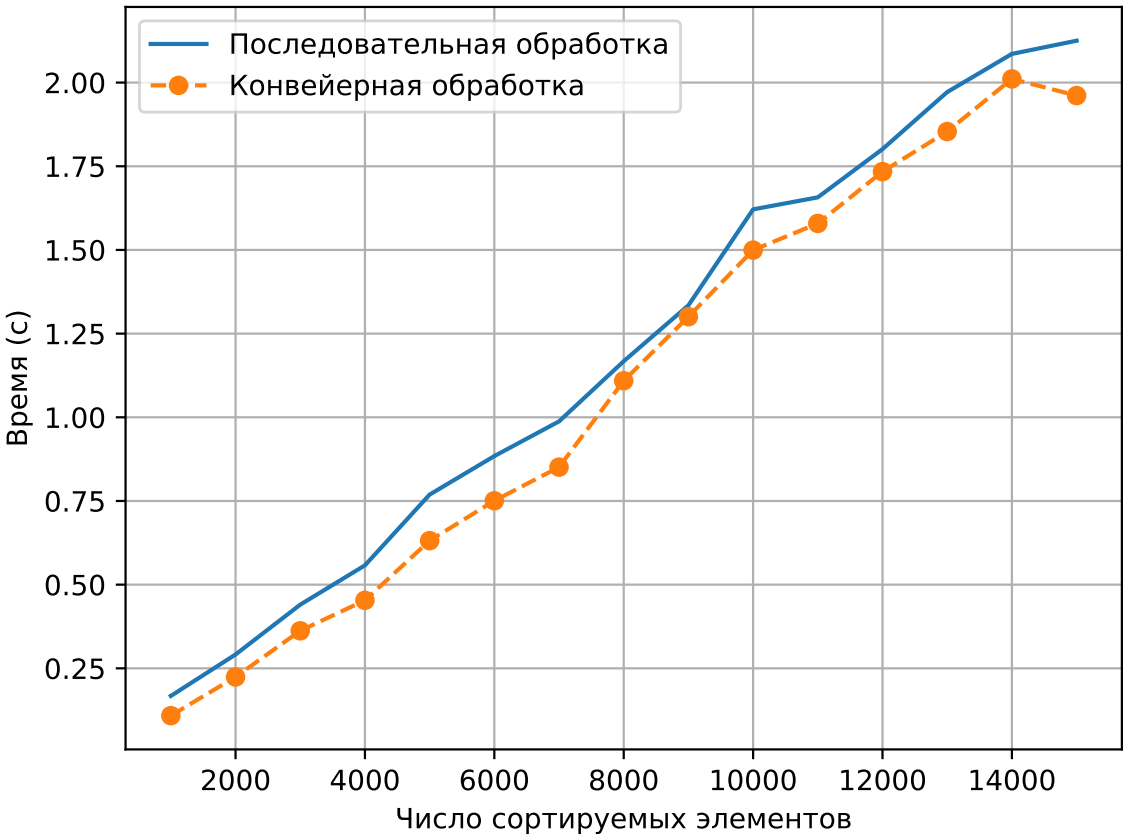

Рисунок 4.3 – Сравнение реализаций обработок заявок по времени получения результата в зависимости от числа элементов в массиве

Вывод

В результате анализа таблицы 4.1, можно сделать вывод что конвейерная обработка заявок позволяет быстрее получать результат: при 110 заявках, при использовании реализации с конвейером данные были получены за 1.41 сек, что в 1.08 раз быстрее, чем при использовании последовательной обработки.

При увеличении числа сортируемых элементов, реализация конвейерной обработки быстрее получает результат: при 15000 сортируемых элементах результат был получен за 1.96 секунд, что в 1.09 раз быстрее чем последовательная обработка.

ЗАКЛЮЧЕНИЕ

В результате исследования было определено, что при использовании реализации конвейерной обработки заявок время получения результата сократилось, относительно последовательной обработки заявок при увеличении числа заявок и при увеличении числа элементов массива. При обработке 110 заявок при 9000 сортируемых элементах результат был получен в 1.08 раз быстрее, при обработке 100 заявок при 15000 сортируемых элементов, результат был получен в 1.09 раз быстрее, данный результат был получен ввиду параллельной обработки заявок на различных этапах различными потоками.

Поставленная цель: получение навыков организации конвейерной обработки данных, была достигнута.

Для поставленной цели были выполнены все поставленные задачи:

1. описать организацию конвейерной обработки данных;
2. описать алгоритмы обработки данных, которые будут использоваться в текущей лабораторной работе;
3. реализовать программу, выполняющую конвейерную обработку с количеством лент не менее трех и программу обрабатывающую заявки последовательно;
4. провести сравнительный анализ времени работы реализаций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Принципы конвейерной технологии [Электронный ресурс]. — Режим доступа: https://www.sites.google.com/site/shoradimon/18-principy-konvejernoj-tehnologii (дата обращения: 24.12.2023).
2. Конвейерная обработка данных [Электронный ресурс]. — Режим доступа: https://studfile.net/preview/1083252/page:25/ (дата обращения: 24.12.2023).
3. Merge sort [Электронный ресурс]. — Режим доступа: https://nauchniestati.ru/spravka/algoritm-sortirovki-sliyaniem (дата обращения: 06.12.2023).
4. clock(3) — Linux [Электронный ресурс]. — Режим доступа: https://man7.org/linux/man-pages/man3/clock.3.html (дата обращения: 28.09.2023).
5. class queue [Электронный ресурс]. — Режим доступа: https://learn.microsoft.com/ru-ru/cpp/standard-library/queue-class (дата обращения: 24.12.2023).
6. Windows 10 Pro 2h21 64-bit [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/ru-ru/software-download/windows10 (дата обращения: 28.09.2023).
7. Что такое WSL [Электронный ресурс]. — Режим доступа: https://learn.microsoft.com/ru-ru/windows/wsl/about (дата обращения: 28.09.2023).

ПРИЛОЖЕНИЕ А

Листинг А.1 – Реализация конвейерной обработки заявок

```
void StartConveyorAsync(std::queue<Request>& start,
   std::queue<Request>& end)
{
    AtomicQueue<Request> secondQ;
    AtomicQueue<Request> thirdQ;

    std::thread t1(std::bind(handle_first, std::ref(start),
       std::ref(secondQ)));
    std::thread t2(std::bind(handle_second, std::ref(secondQ),
       std::ref(thirdQ)));
    std::thread t3(std::bind(handle_third, std::ref(thirdQ),
       std::ref(end)));
    t1.join();
    t2.join();
    t3.join();
}
```

Листинг А.2 – Реализация первого этапа обработки заявок (сортировка копии массива)

```
void handle_first(std::queue<Request>& from,
   AtomicQueue<Request>& to)
{
    bool isWorking = true;
    while (isWorking)
    {
        if (from.size() > 0)
        {
            timespec start, end;
            Request curRequest = from.front();

            if (curRequest.is_last)
                isWorking = false;

            from.pop();
            std::vector<int> copy = curRequest.req;
            start = getTime();
            mergeSort(copy, 0, copy.size() - 1);
```

```
18              end = getTime();
19
20              curRequest.time_start_1 = start;
21              curRequest.time_end_1 = end;
22              to.push(curRequest);
23          }
24
25      }
26
27  }
```

Листинг А.3 – Реализация второго этапа обработки заявок (сортировка массива с использование дополнительных потоков)

```
1  void handle_second(AtomicQueue<Request>& from,
       AtomicQueue<Request>& to)
2  {
3      bool isWorking = true;
4      while (isWorking)
5      {
6          if (from.size() > 0)
7          {
8              timespec start{}, end{};
9              Request curRequest = from.front();
10             from.pop();
11             if (curRequest.is_last)
12                 isWorking = false;
13
14             start = getTime();
15             mergeSortMultiThread(curRequest.req, 0,
                   curRequest.req.size() - 1, 4);
16             end = getTime();
17             curRequest.time_start_2 = start;
18             curRequest.time_end_2 = end;
19             to.push(curRequest);
20         }
21     }
22 }
```

Листинг А.4 – Реализация третьего этапа обработки заявок (запись отсортированного массива в файл)

```
1  void handle_third(AtomicQueue<Request>& from,
       std::queue<Request>& endQ)
```

```
{

    bool isWorking = true;
    while (isWorking)
    {
        if (from.size() > 0)
        {
            Request curRequest = from.front();
            from.pop();
            if (curRequest.is_last)
                isWorking = false;

            write_into_file(curRequest);
            endQ.push(curRequest);
        }
    }

}
```

Листинг А.5 – Реализация последовательной обработки заявок

```
void Linear(std::queue<Request>& startQ, std::queue<Request>&
   endQ)
{
    while (!startQ.empty())
    {
        timespec start, end;
        Request curRequest = startQ.front();
        startQ.pop();
        std::vector<int> copy = curRequest.req;
        start = getTime();
        mergeSort(copy, 0, copy.size() - 1);
        end = getTime();
        curRequest.time_start_1 = start;
        curRequest.time_end_1 = end;


        start = getTime();
        mergeSortMultiThread(curRequest.req, 0,
           curRequest.req.size() - 1, 4);
        end = getTime();
        curRequest.time_start_2 = start;
        curRequest.time_end_2 = end;
```

```
21          write_into_file(curRequest);
22          endQ.push(curRequest);
23      }
24  }
```

Листинг А.6 – Реализация очереди с атомарным добавлением, удалением и получением элемента и длинны

```
1  template<class T>
2  class AtomicQueue
3  {
4   public:
5      void push(const T& value)
6      {
7          std::lock_guard<std::mutex> lock(m_mutex);
8          m_queque.push(value);
9      }
10
11     void pop()
12     {
13         std::lock_guard<std::mutex> lock(m_mutex);
14         m_queque.pop();
15     }
16     T front()
17     {
18         std::lock_guard<std::mutex> lock(m_mutex);
19         return m_queque.front();
20     }
21     size_t size()
22     {
23         std::lock_guard<std::mutex> lock(m_mutex);
24         return m_queque.size();
25     }
26   public:
27     std::queue<T> m_queque;
28     mutable std::mutex m_mutex;
29 };
```