



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 рубежному контролю № 1  
по курсу «Анализ алгоритмов»

Студент ИУ7-54Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

Булдаков М.  
(И. О. Фамилия)

Преподаватель

\_\_\_\_\_  
(Подпись, дата)

Волкова Л. Л.  
(И. О. Фамилия)

2023 г.

# СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ</b>	<b>3</b>
<b>1 Аналитический раздел</b>	<b>4</b>
1.1 Многопоточность . . . . .	4
1.2 Исправления орфографических ошибок в тексте . . . . .	4
1.3 Использование потоков для исправления орфографических оши- бок . . . . .	5
<b>2 Конструкторский раздел</b>	<b>6</b>
2.1 Требования к программному обеспечению . . . . .	6
2.2 Описание используемых типов данных . . . . .	6
2.3 Разработка алгоритмов . . . . .	7
<b>3 Технологический раздел</b>	<b>11</b>
3.1 Средства реализации . . . . .	11
3.2 Сведения о модулях программы . . . . .	12
3.3 Реализация алгоритмов . . . . .	12
3.4 Функциональные тесты . . . . .	21
<b>4 Исследовательский раздел</b>	<b>22</b>
4.1 Демонстрация работы программы . . . . .	22
<b>ЗАКЛЮЧЕНИЕ</b>	<b>25</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>	<b>26</b>

# ВВЕДЕНИЕ

С развитием вычислительных систем появилась потребность в параллельной обработке данных для повышения эффективности систем, ускорения вычислений и более рационального использования имеющихся ресурсов. Благодаря совершенствованию процессоров стало возможно использовать их для выполнения множества одновременных задач, что привело к появлению понятия «многопоточность» [1].

Задание рубежного контроля — многопоточное исправление орфографических ошибок.

Цель данного рубежного контроля — описать принципы параллельных вычислений на основе нативных потоков для исправления орфографических ошибок в тексте. Для достижения поставленной цели необходимо выполнить следующие задачи:

- описать алгоритм исправления орфографических ошибок в тексте;
- спроектировать программное обеспечение, реализующее алгоритм и его параллельную версию;
- выбрать инструменты для реализации и замера процессорного времени выполнения реализаций алгоритмов.

# 1 Аналитический раздел

В данном разделе будет представлена информация о многопоточности и исследуемом алгоритме исправления орфографических ошибок в тексте.

## 1.1 Многопоточность

Многопоточность — это способность центрального процессора одновременно выполнять несколько потоков, используя ресурсы одного процессора. Каждый поток представляет собой последовательность инструкций, которые могут выполняться параллельно с другими потоками, созданными одним и тем же процессом [2] .

Процессом называют программу в стадии выполнения [3]. Один процесс может иметь один или несколько потоков. Поток — это часть процесса, которая выполняет задачи, необходимые для выполнения приложения. Процесс завершается, когда все его потоки полностью завершены.

Одной из сложностей, связанных с использованием потоков, является проблема доступа к данным. Основным ограничением является невозможность одновременной записи в одну и ту же ячейку памяти из разных потоков. Это означает, что нужен механизм синхронизации доступа к данным, так называемый «мьютекс» (от англ. mutex — mutual exclusion, взаимное исключение). Мьютекс может быть захвачен одним потоком для работы в режиме монопольного использования или освобожден. Если два потока попытаются захватить мьютекс одновременно, то успех будет у одного потока, а другой будет блокирован, пока мьютекс не освободится.

## 1.2 Исправления орфографических ошибок в тексте

Для распознавания слов, написанных с ошибками, используется расстояние Левенштейна — минимальное количество ошибок, исправление которых приводит одно слово к другому [4]. Т. о. для введенного слова осуществляется проверка по корпусу, если данное слово не найдено в корпусе, то ищется ближайшее слово к данному по расстоянию Левенштейна.

Кроме того, следует вводить ограничение на количество ошибок, которые допускается допустить. Как говорит поговорка: «Если в слове хлеб допустить всего четыре ошибки, то получится слово пиво» [4]. Если фиксируется число

ошибок, то для коротких слов оно может оказаться избыточным. Верхнюю границу числа ошибок обычно ограничивают как процентным соотношением, так и фиксированным числом. Например, не более 30% букв входного слова, но не более 3. При этом все равно стараются найти слова с минимальным количеством ошибок [4].

### **1.3 Использование потоков для исправления орфографических ошибок**

Поскольку задача сводится к поиску слова в корпусе, можно распараллелить поиск по этому корпусу. В таком случае каждый поток будет вычислять расстояние Левенштейна между заданным словом и некоторым словом из корпуса и в случае, если расстояние будет удовлетворять требованиям, то данное слово будет записано в массив. Для определения наилучших соответствий необходимо хранить минимальное количество ошибок на текущий момент, т. е. возможна ситуация, когда в одном потоке минимальное количество ошибок будет считано, а в другом в тот же момент изменено, следовательно возникает конфликт. То же касается и записи подходящих слов в массив, требуется отбирать лучшие  $k$  слов, поэтому возможна ситуация, когда значение длины массива считывается в одном потоке и в тот же момент изменяется в другом потоке, т. е. возникает конфликт. Для решения проблем синхронизации необходимо использовать мьютекс, чтобы обеспечить монополярный доступ к длине массива и текущему минимальному количеству ошибок.

Корпус можно представить с помощью сегментированного словаря. Каждый сегмент будет хранить слова, которые начинаются на одну и ту же букву алфавита. Поскольку ошибки чаще бывают в середине слова, а не в начале, то такой подход позволит минимизировать количество слов среди которых нужно осуществлять поиск.

### **Вывод**

В данном разделе была представлена информация о многопоточности и исследуемом алгоритме.

## 2 Конструкторский раздел

В этом разделе будет представлено описание используемых типов данных, а также схемы алгоритмов исправления орфографических ошибок.

### 2.1 Требования к программному обеспечению

Программа должна поддерживать два режима работы: режим массового замера времени и режим исправления введенного слова.

Режим массового замера времени должен обладать следующей функциональностью:

- генерировать корпус слов;
- осуществлять массовый замер, используя сгенерированные данные;
- результаты массового замера должны быть представлены в виде таблицы.

К режиму исправления введенного слова выдвигается следующий ряд требований:

- возможность вводить слова, которые отсутствуют в корпусе;
- слова вводятся на кириллице;
- наличие интерфейса для выбора действий;
- на выходе программы, набор из самых близких слов к введенному.

### 2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры и типы данных:

- слово — массив букв;
- корпус — массив слов, отсортированный в лексикографическом порядке;
- сегментированный словарь — словарь, где ключом является буква, а значением корпус;
- мьютекс — примитив синхронизации.

## 2.3 Разработка алгоритмов

На рисунке 2.1 представлена схема поиска ближайших слов в словаре, алгоритм одинаков в случае если используются потоки и если не используются, поскольку потоки применяются только для поиска слов в корпусе. На рисунке 2.2 представлена схема поиска ближайших слов в корпусе без использования потоков. На рисунке 2.3 представлена схема поиска ближайших слов в корпусе с использованием потоков. На рисунке 2.4 представлена схема алгоритма программы, выполняющейся в потоке.

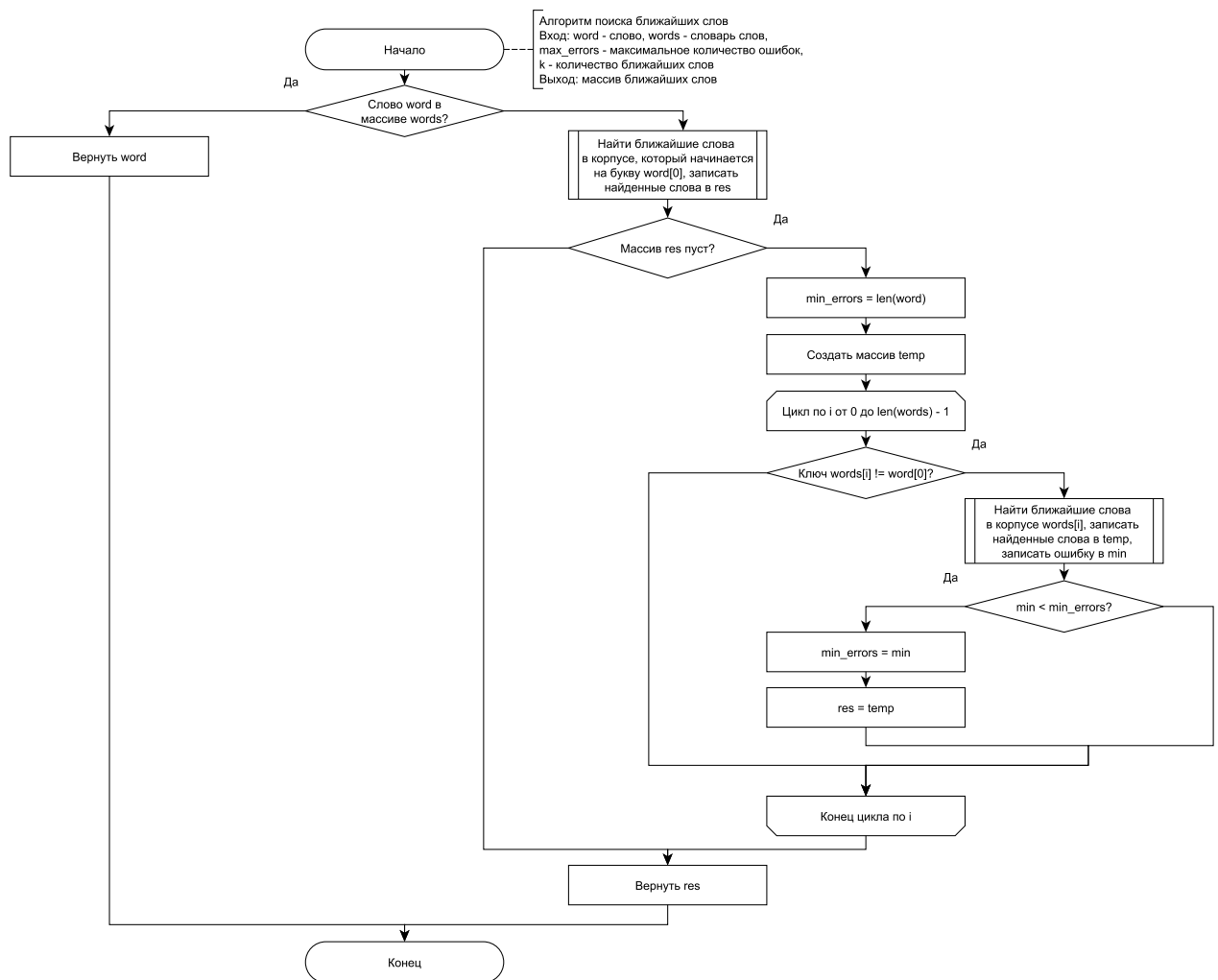


Рисунок 2.1 – Схема алгоритма поиска ближайших слов в сегментированном словаре

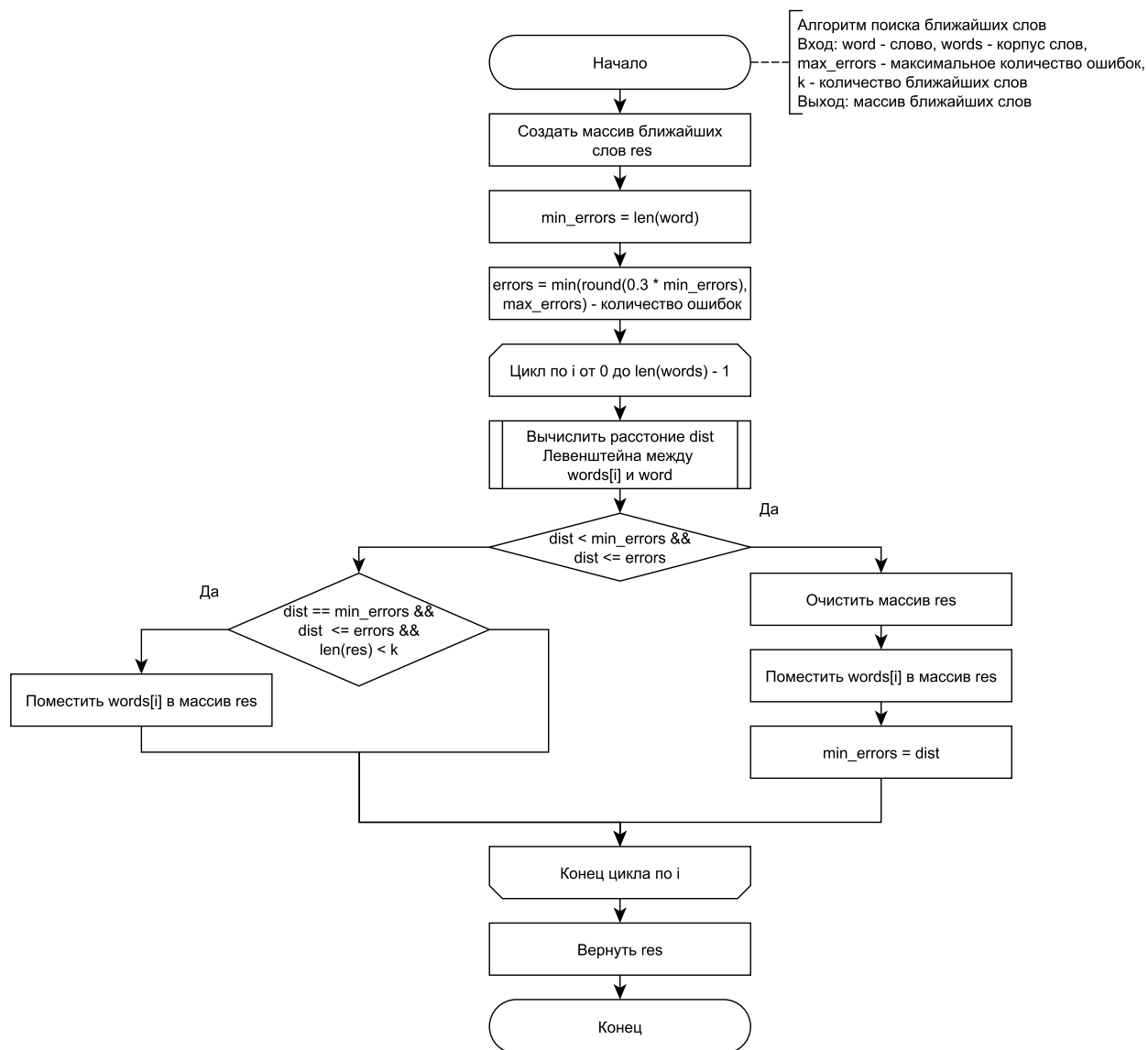


Рисунок 2.2 – Схема алгоритма поиска ближайших слов в корпусе



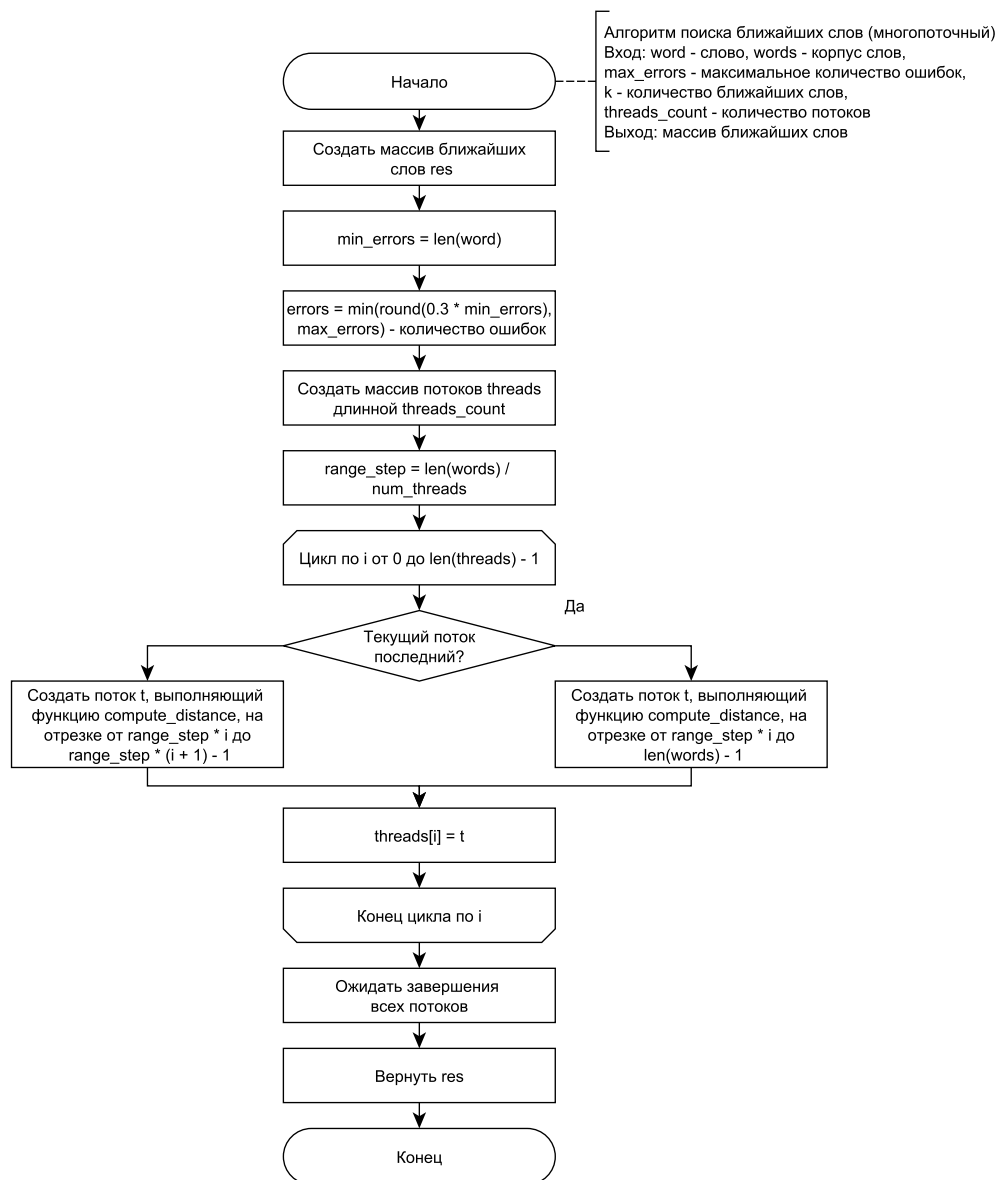


Рисунок 2.3 – Схема многопоточного алгоритма поиска ближайших слов в корпусе

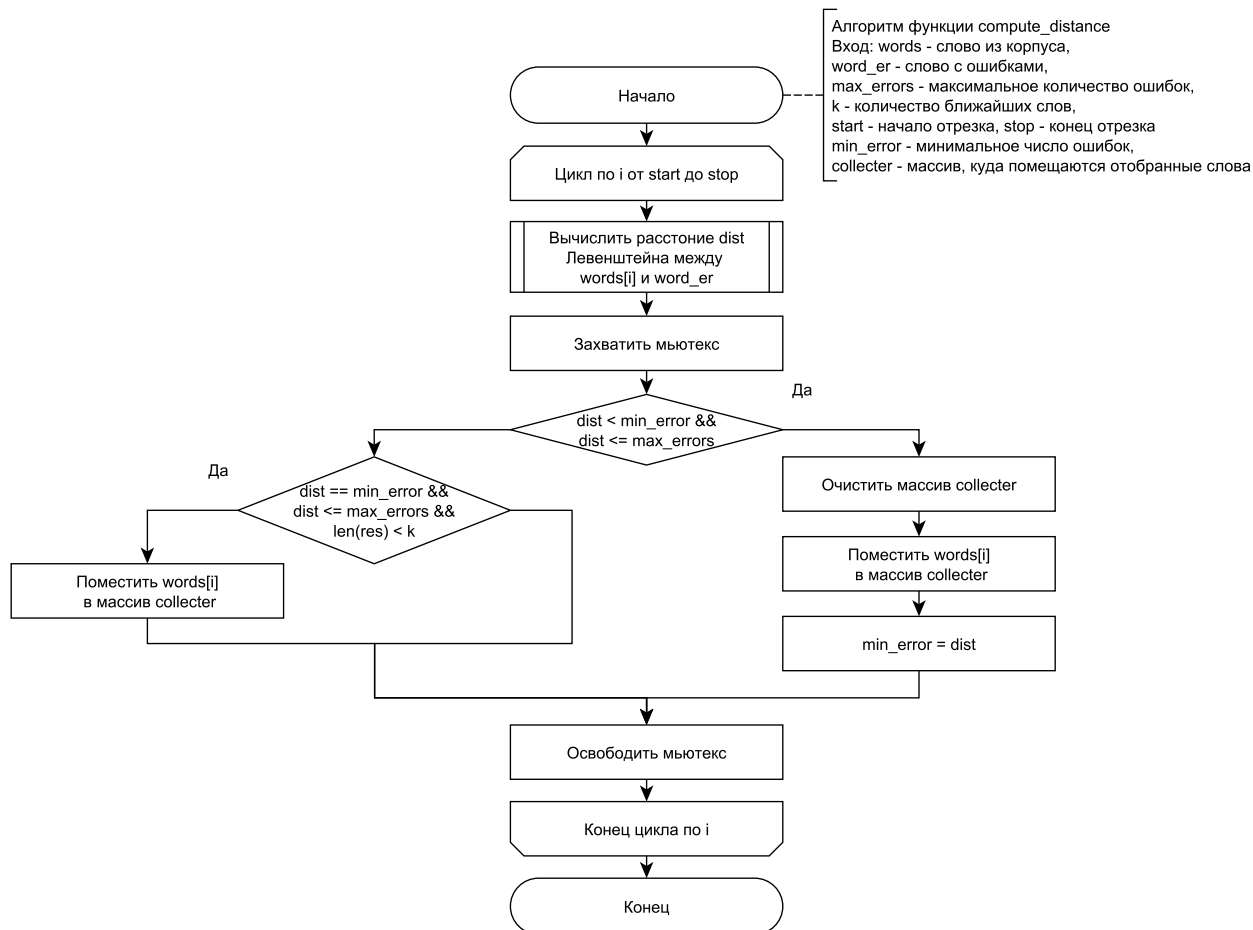


Рисунок 2.4 – Схема алгоритма потока

Поскольку в массиве в каждый момент времени содержатся слова, которые находятся на одинаковом расстоянии от введенного слова, то нет необходимости в какой-либо сортировке результатов, т. к. любое из найденных слов с одинаковой вероятностью может оказаться искомым. Поэтому шаг особого слияния результатов не требуется и достаточно помещать подходящее слово в массив.

## Вывод

На основе теоретических данных, полученных из аналитического раз-  
дела были построены схемы требуемых алгоритмов.

## 3 Технологический раздел

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и функциональные тесты.

### 3.1 Средства реализации

Для реализации данной работы был выбран язык *C++* [5]. Данный выбор обусловлен следующим:

- язык поддерживает все структуры данных, которые выбраны в результате проектирования;
- язык позволяет реализовать все алгоритмы, выбранные в результате проектирования;
- язык позволяет работать с нативными потоками [6].

Время выполнения реализаций было измерено с помощью функции *clock* [7]. Для хранения слов использовалась структура данных *wstring* [8], в качестве массивов использовалась структура данных *vector* [9], в качестве словаря использовалась структура данных *map* [10]. В качестве примитива синхронизации использовался *mutex* [11].

Для создания потоков и работы с ними был использован класс *thread* из стандартной библиотеки выбранного языка [6]. В листинге 3.1, приведен пример работы с описанным классом, каждый объект класса представляет собой поток операционной системы, что позволяет нескольким функциям выполняться параллельно [6].

### Листинг 3.1 – Пример работы с классом `thread`

```
1 #include <iostream>
2 #include <thread>
3
4 void foo(int a)
5 {
6     std::cout << a << '\n';
7 }
8
9 int main()
10 {
11     std::thread thread(foo, 10);
12     thread.join();
13
14     return 0;
15 }
```

## 3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

- *main.cpp* — файл, содержащий функцию *main*;
- *correcter.cpp* — файл, содержащий код реализаций всех алгоритмов исправления ошибок;
- *measure\_time.cpp* — файл, в котором содержатся функции для замера и вывода времени выполнения реализаций алгоритмов;
- *utils.cpp* — файл, в котором содержатся вспомогательные функции;
- *levenstein.cpp* — файл, в котором содержится реализация алгоритма поиска расстояния Левенштейна.

## 3.3 Реализация алгоритмов

На листингах 3.2 – 3.5 приведены реализации функций, который осуществляют работу с сегментированным словарем. В листинге 3.6 приведена реализация алгоритма поиска слов в корпусе. В листинге 3.7 приведена реализация алгоритма поиска слов в корпусе с использованием дополнительных

потоков. В листинге 3.8 приведена реализация функции, которая выполняется потоком.

Листинг 3.2 – Функция поиска слов в словаре с дополнительными потоками

```
1 std::vector<std::wstring> get_closest_words_with_segmented_mt(  
2     std::map<wchar_t, std::vector<std::wstring>> &words,  
3     const std::wstring &word, size_t k, size_t max_errors,  
4         size_t num_threads){  
5  
6     if (is_word_in_vec(words[word[0]], word))  
7         return {word};  
8  
9     auto res = get_closest_words_mt(words[word[0]], word, k,  
10         max_errors, num_threads);  
11  
12     std::vector<std::wstring> ans = res.first;  
13  
14     if (ans.empty()){  
15         ans = full_search_mt(words, word, k, max_errors,  
16             num_threads);  
17     }  
18  
19     return ans;  
20 }
```

### Листинг 3.3 – Функция поиска слов в словаре

```
1 std::vector<std::wstring> get_closest_words_with_segmented(  
2     std::map<wchar_t, std::vector<std::wstring>> &words,  
3     const std::wstring &word, size_t k, size_t max_errors){  
4  
5     if (is_word_in_vec(words[word[0]], word))  
6         return {word};  
7  
8     auto res = get_closest_words(words[word[0]], word, k,  
9         max_errors);  
10  
11     std::vector<std::wstring> ans = res.first;  
12  
13     if (ans.empty()){  
14         ans = full_search(words, word, k, max_errors);  
15     }  
16  
17     return ans;  
18 }
```

### Листинг 3.4 – Функция полного перебора словаря

```
1 std::vector<std::wstring> full_search(  
2     std::map<wchar_t, std::vector<std::wstring>> &words,  
3     const std::wstring &word, size_t k, size_t max_errors){  
4  
5     std::vector<std::wstring> res;  
6     size_t min = word.size();  
7  
8     for (const auto &p: words){  
9         if (p.first != word[0]){  
10             auto temp = get_closest_words(p.second, word, k,  
11                 max_errors);  
12  
13             if (temp.second < min){  
14                 res = temp.first;  
15                 min = temp.second;  
16             }  
17         }  
18     }  
19     return res;  
20 }
```



Листинг 3.5 – Функция полного перебора словаря с дополнительными потоками

```
1 std::vector<std::wstring> full_search_mt(  
2     std::map<wchar_t, std::vector<std::wstring>> &words,  
3     const std::wstring &word, size_t k, size_t max_errors,  
4         size_t num_threads){  
5  
6     std::vector<std::wstring> res;  
7     size_t min = word.size();  
8  
9     for (const auto &p: words){  
10         if (p.first != word[0]){  
11             auto temp = get_closest_words_mt(p.second, word, k,  
12                 max_errors, num_threads);  
13  
14             if (temp.second < min){  
15                 res = temp.first;  
16                 min = temp.second;  
17             }  
18         }  
19     }  
20     return res;  
}
```

### Листинг 3.6 – Функция исправления ошибок

```
1 std::pair<std::vector<std::wstring>, size_t> get_closest_words(  
2     const std::vector<std::wstring> &words, const std::wstring  
3     &word,  
4     size_t k, size_t max_errors) {  
5  
6     std::vector<std::wstring> temp;  
7     size_t min = word.size();  
8  
9     size_t errors = std::min(static_cast<size_t>(std::ceil(0.3 *  
10         word.size()))), max_errors);  
11  
12     for (const auto &cur_word: words) {  
13         int dist = lev_mtr(cur_word, word);  
14         if (dist < min && dist <= errors) {  
15             temp.clear();  
16             temp.push_back(cur_word);  
17             min = dist;  
18         } else if (dist == min && dist <= errors && temp.size()  
19             < k)  
20             temp.push_back(cur_word);  
21     }  
  
22     return {temp, min};  
23 }
```

### Листинг 3.7 – Функция многопоточного исправления ошибок

```

1  std::pair<std::vector<std::wstring>, size_t>
    get_closest_words_mt(
2      const std::vector<std::wstring> &words, const std::wstring
        &word,
3      size_t k, size_t max_errors, size_t num_threads) {
4
5      size_t min = word.size();
6      size_t errors = std::min(static_cast<size_t>(std::ceil(0.3 *
        word.size()))), max_errors);
7      std::vector<std::wstring> collector;
8      std::thread threads[num_threads];
9
10     size_t range_step = words.size() / num_threads;
11
12     for (size_t i = 0; i < num_threads; ++i) {
13         if (i != num_threads - 1) {
14             threads[i] = std::thread(
15                 compute_distance, words,
16                 word, errors, k,
17                 range_step * i, range_step * (i + 1),
18                 std::ref(min), std::ref(collector));
19         } else
20             threads[i] = std::thread(
21                 compute_distance, words,
22                 word, errors, k,
23                 range_step * i, words.size(),
24                 std::ref(min), std::ref(collector));
25     }
26
27     for (size_t i = 0; i < num_threads; ++i) {
28         threads[i].join();
29     }
30
31     return {collector, min};
32 }

```

Листинг 3.8 – Функция, выполняющаяся в потоке

```
1 std::mutex mutex;
2
3 void compute_distance(const std::vector<std::string> &words,
4                      const std::string &word_er,
5                      size_t errors,
6                      size_t k,
7                      size_t start,
8                      size_t stop,
9                      size_t &min,
10                     std::vector<std::string> &collector) {
11     for (size_t i = start; i < stop; ++i) {
12         const auto word_cor = words[i];
13         int dist = lev_mtr(word_cor, word_er);
14
15         mutex.lock();
16         if (dist < min && dist <= errors) {
17             collector.clear();
18             collector.push_back(word_cor);
19             min = dist;
20         } else if (dist == min && dist <= errors &&
21                  collector.size() < k)
22             collector.push_back(word_cor);
23         mutex.unlock();
24     }
25 }
```

### 3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для разработанных алгоритмов исправления ошибок. Для данных тестов максимальное количество ошибок равно двум и из массива выбираются 3 слова. Все тесты пройдены успешно. В таблице 3.1 пустое слово обозначается с помощью  $\lambda$ .

Таблица 3.1 – Функциональные тесты

Корпус	Слово	Ожидаемый результат
[мама, мыла, раму]	мама	[мама]
[мама, мыла, раму]	мамы	[мама]
[мама, мыла, раму]	мыма	[мама, мыла]
[мама, мыла, раму]	ахтунг	[ ]
[ ]	ахтунг	[ ]
[ ]	$\lambda$	[ ]
[Мама, Мыла, Раму]	$\lambda$	[ ]

### Вывод

Были разработаны и протестированы спроектированные алгоритмы исправления ошибок.

## 4 Исследовательский раздел

В данном разделе будут приведены: пример работы программы, постановка исследования и сравнительный анализ алгоритмов на основе полученных данных.

### 4.1 Демонстрация работы программы

На рисунках 4.1 и 4.2 представлена демонстрация работы разработанного программного обеспечения, а именно показаны результаты исправления ошибок в слове *сламандры* и *маламандры*.

```
Меню:
1 - Прочитать корпус
2 - Проверить слово (без потоков)
3 - Проверить слово (с потоками)
4 - Вывести статистику корпуса
0 - Выйти

Выберите пункт меню:2
2
Введите слово:сламандры
сламандры
Максимальное число ошибок:3
3
Сколько слов вывести:3
3
Результат:
саламандр
саламандры
Меню:
1 - Прочитать корпус
2 - Проверить слово (без потоков)
3 - Проверить слово (с потоками)
4 - Вывести статистику корпуса
0 - Выйти

Выберите пункт меню:|
```

Рисунок 4.1 – Демонстрация работы программы при исправлении слова без потоков

```
Меню:
1 - Прочитать корпус
2 - Проверить слово (без потоков)
3 - Проверить слово (с потоками)
4 - Вывести статистику корпуса
0 - Выйти

Выберите пункт меню:3
3
Введите слово:маламандры
маламандры
Максимальное число ошибок:3
3
Сколько слов вывести:3
3
Количество потоков:3
3
Результат:
саламандр
саламандры
Меню:
1 - Прочитать корпус
2 - Проверить слово (без потоков)
3 - Проверить слово (с потоками)
4 - Вывести статистику корпуса
0 - Выйти

Выберите пункт меню:|
```

Рисунок 4.2 – Демонстрация работы программы при исправлении слова с дополнительными потоками



## ЗАКЛЮЧЕНИЕ

Цель данной лабораторной работы была достигнута, а именно были описаны принципы параллельных вычислений на основе нативных потоков для исправления орфографических ошибок в тексте.

Для достижения поставленной цели были выполнены следующие задачи:

- 1) описан алгоритм исправления орфографических ошибок в тексте;
- 2) спроектировано программное обеспечение, реализующее алгоритм и его параллельную версию;
- 3) выбраны инструменты для реализации и замера процессорного времени выполнения реализаций алгоритмов.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Гладышев Е., Мурыгин А.* Многопоточность в приложениях // Актуальные проблемы авиации и космонавтики. — 2012. — № 8.
2. *Stoltzfus J.* Multithreading [Электронный ресурс]. — Режим доступа: <https://www.techopedia.com/definition/24297/multithreading-computer-architecture> (дата обращения: 07.12.2023).
3. *Ричард Стивенс У., Стивен Раго А.* UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018. — С. 994.
4. Автоматическая обработка текстов на естественном языке и компьютерная лингвистика / Е. Большакова [и др.] //. — М.: МИЭМ, 2011. — С. 122—124.
5. C++ reference [Электронный ресурс]. — Режим доступа: <https://en.cppreference.com/w/> (дата обращения: 20.12.2023).
6. Concurrency support library [Электронный ресурс]. — Режим доступа: <https://en.cppreference.com/w/cpp/thread> (дата обращения: 20.12.2023).
7. std::clock [Электронный ресурс]. — Режим доступа: <https://en.cppreference.com/w/cpp/chrono/c/clock> (дата обращения: 20.12.2023).
8. Strings library [Электронный ресурс]. — Режим доступа: <https://en.cppreference.com/w/cpp/string> (дата обращения: 20.12.2023).
9. std::vector [Электронный ресурс]. — Режим доступа: <https://en.cppreference.com/w/cpp/string> (дата обращения: 20.12.2023).
10. std::map [Электронный ресурс]. — Режим доступа: <https://en.cppreference.com/w/cpp/container/map> (дата обращения: 22.12.2023).
11. std::mutex [Электронный ресурс]. — Режим доступа: <https://en.cppreference.com/w/cpp/thread/mutex> (дата обращения: 20.12.2023).