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 вычисленийs»

Студент группы ИУ7-54Б _____ (Подпись, дата) Спирин М.П. (Фамилия И.О.)

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

Преподаватель _____ (Подпись, дата) Строганов Ю. В. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

По мере развития вычислительных систем программисты столкнулись с необходимостью производить параллельную обработку данных для улучшения отзывчивости системы и ускорения производимых вычислений. Благодаря развитию процессоров стало возможным использовать один процессор для выполнения нескольких параллельных операций, что дало начало термину «Многопоточность».

Целью данной лабораторной работы является получение навыков организации параллельного выполнения операций.

Для поставленной цели необходимо выполнить следующие задачи.

1) Описать основы распараллеливания вычислений.

2) Разработать программное обеспечение, которое реализует однопоточный алгоритм нахождения всех вхождений подстроки в файле.

3) Разработать и реализовать многопоточную версию данного алгоритма.

4) Определить средства программной реализации.

5) Выполнить замеры реального времени, затрачиваемого на работу реализации алгоритма.

6) Сравнить по времени работы версии реализации алгоритма.

1 Аналитическая часть

В данном разделе будет представлена информация о многопоточности и реализуемом алгоритме нахождения всех вхождений подстроки в файле.

1.1 Многопоточность

Многопоточность (англ. *multithreading*) — это способность центрального процессора (ЦП) обеспечивать одновременное выполнение нескольких потоков в рамках использования ресурсов одного процессора [1]. Поток — последовательность инструкций, которая может исполняться параллельно с другими потоками одного и того же породившего их процесса.

Процессом называют программу в ходе своего выполнения [2]. Таким образом, когда запускается программа, создается процесс. Один процесс может состоять из одного и более потоков. Таким образом, поток является сегментом процесса, сущностью, которая выполняет задачи, стоящие перед исполняемым приложением. Процесс завершается, когда все его потоки заканчивают работу. Каждый поток в операционной системе является задачей, которую должен выполнить процессор.

При написании программы, использующей несколько потоков, следует учесть, что не всегда есть выигрыш от использования нескольких потоков. Необходимо создавать потоки для независимых по данным и выполнять их параллельно, тем самым сокращая общее время выполнения процесса.

Одной из проблем, появляющихся при использовании потоков, является проблема совместного доступа к информации. Фундаментальным ограничением является запрет на запись из двух и более потоков в одну ячейку памяти одновременно.

Из этого следует, что необходим примитив синхронизации обращения к данным – так называемый мьютекс (англ. *mutex — mutual exclusion*). Он может быть захвачен для работы в монопольном режиме или освобожден, чтобы позволить следующему потокому его захватить. Так, если 2 потока одновременно пытаются захватить мьютекс, успевает только один, а другой

будет ждать освобождения.

Набор инструкций, выполняемых между захватом и освобождением мьютекса, называется *критической секцией*. Поскольку в то время, пока мьютекс захвачен, остальные потоки, требующие выполнения критической секции для доступа к одним и тем же данным, ждут освобождения мьютекса для его последующего захвата, требуется разрабатывать программное обеспечение таким образом, чтобы критическая секция была минимальной по объему.

1.2 Поиск подстроки

Поиск подстроки в строке — задача поиска информации. Применяется в виде встроенной функции в текстовых редакторах, СУБД, поисковых машинах, языках программирования и т.д.

В данной лабораторной работе проводится распараллеливание алгоритма поиска всех вхождений искомой подстроки в исходный файл с последующей записью номеров строки и столбца первого символа найденного вхождения в результирующий файл.

Так как последовательное чтение из файла быстрее, чем случайное, в параллельной реализации алгоритма будет использоваться 1 читающий поток [1]. Кроме того, параллельное чтение с жесткого диска возможно только при наличии нескольких головок для чтения, что на практике далеко не всегда доступно.

Считанные строки будут равномерно распределены между несколькими потоками, так как их обработка не зависит от других строк. Именно за счет распараллеливания обработки строк возможно получить какой-то выигрыш в данном алгоритме.

Поскольку одновременная запись в файл из множества потоков невозможна, для записи в файл будет также использоваться только 1 поток.

Вывод

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

2 Конструкторская часть

В данном разделе разработаны схемы реализаций алгоритма поиска всех вхождений подстроки в файле.

2.1 Требования к ПО

К программе предъявлен ряд требований:

— должен присутствовать интерфейс для выбора действий;

— должна работать с нативными потоками;

— должна замерять реальное время, затрачиваемое на работу реализаций алгоритмов.

2.2 Требования к замерам времени

Процессорное время — это время, которое процессор тратит на выполнение задачи или программы. Оно измеряется в тактах процессора.

В данной лабораторной работе необходимо найти реальное время работы программы. Так как в данной работе используется многопоточность, будет возникать необходимость блокировать работу потоков на мьютексах или семафорах. Это время также необходимо учитывать при сравнении многопоточной и однопоточной реализации, несмотря на то, что поток не выполняет никаких действий в это время.

Для этого необходимо замерить реальное время, прошедшее от начала работы главного потока и до конца его выполнения.

2.3 Разработка алгоритмов

На рисунках 2.1 – 2.5 приведены схемы однопоточной и многопоточной версий алгоритма поиска подстроки в файле.

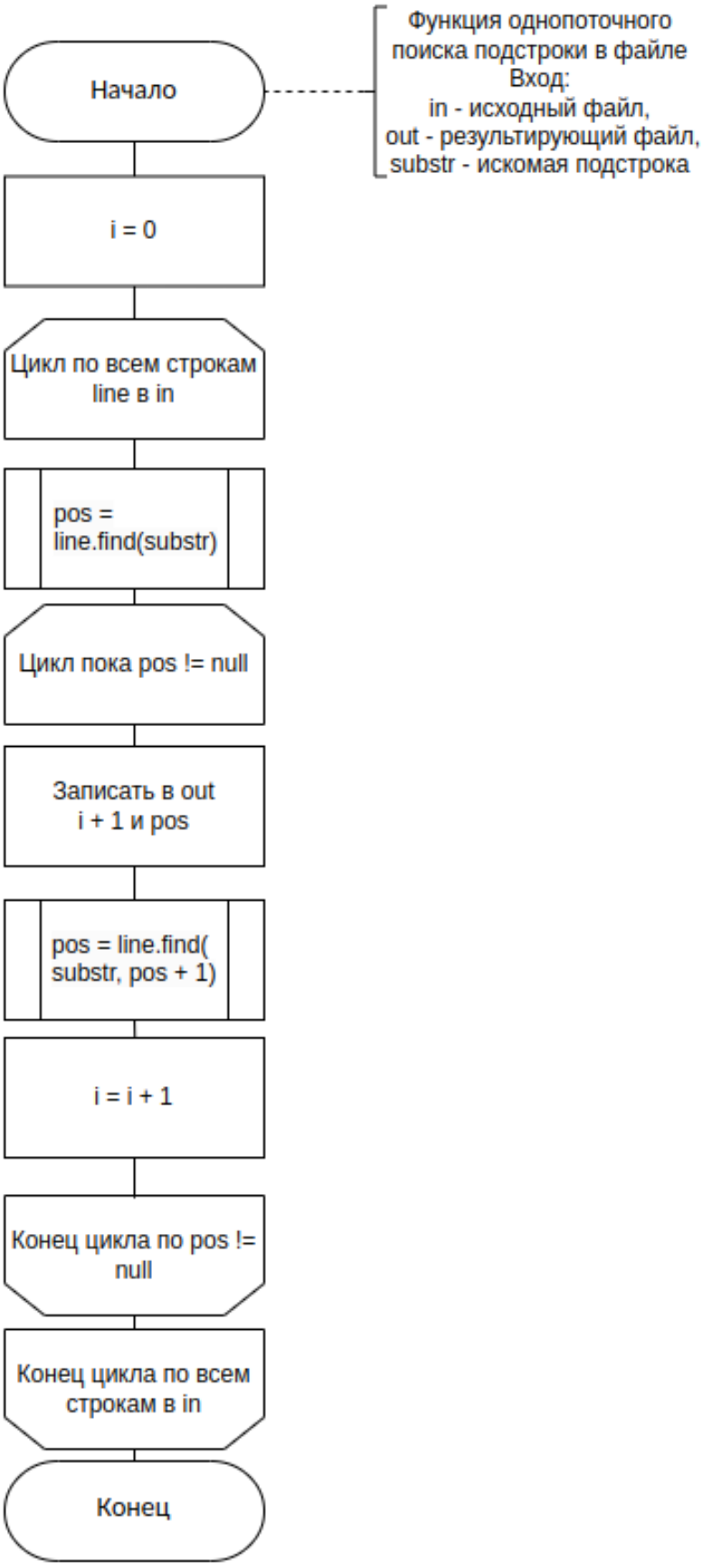

Рисунок 2.1 – Схема однопоточного алгоритма поиска подстроки в файле

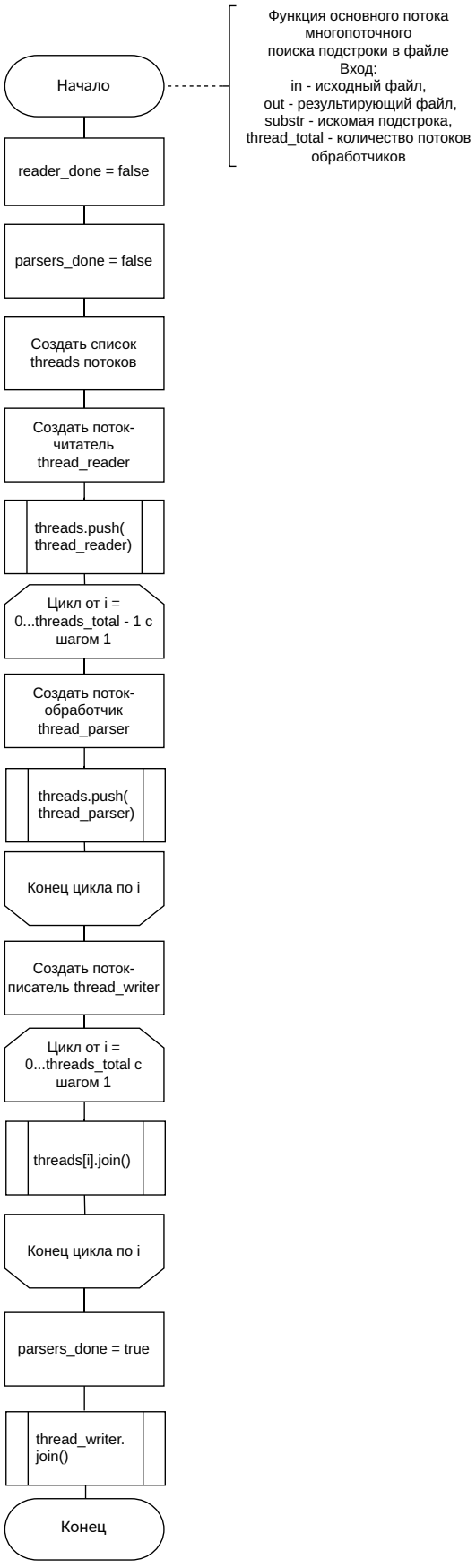

Рисунок 2.2 – Схема алгоритма работы основного потока, запускающего вспомогательные потоки

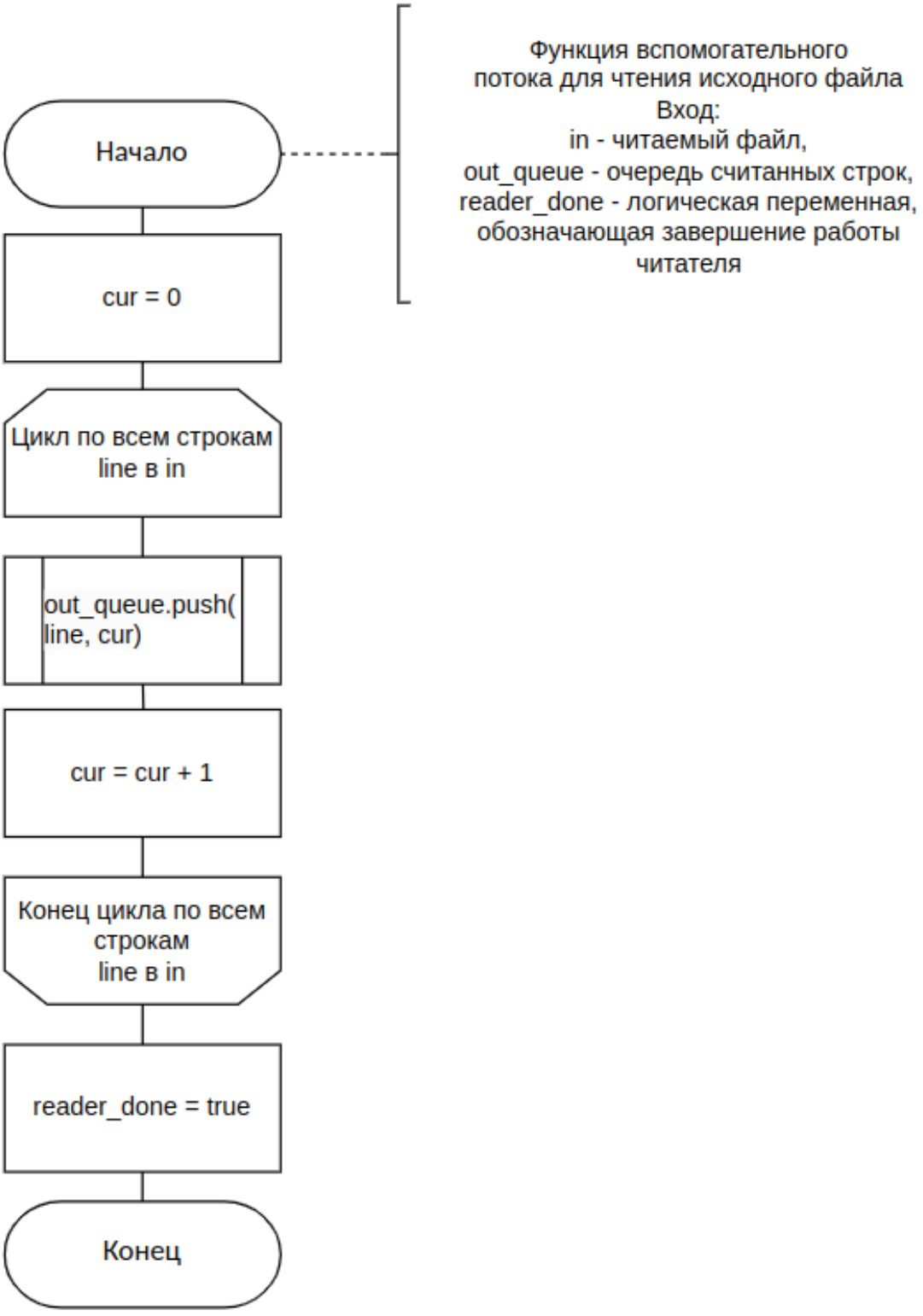

Рисунок 2.3 – Схема алгоритма работы потока, выполняющего чтение из файла

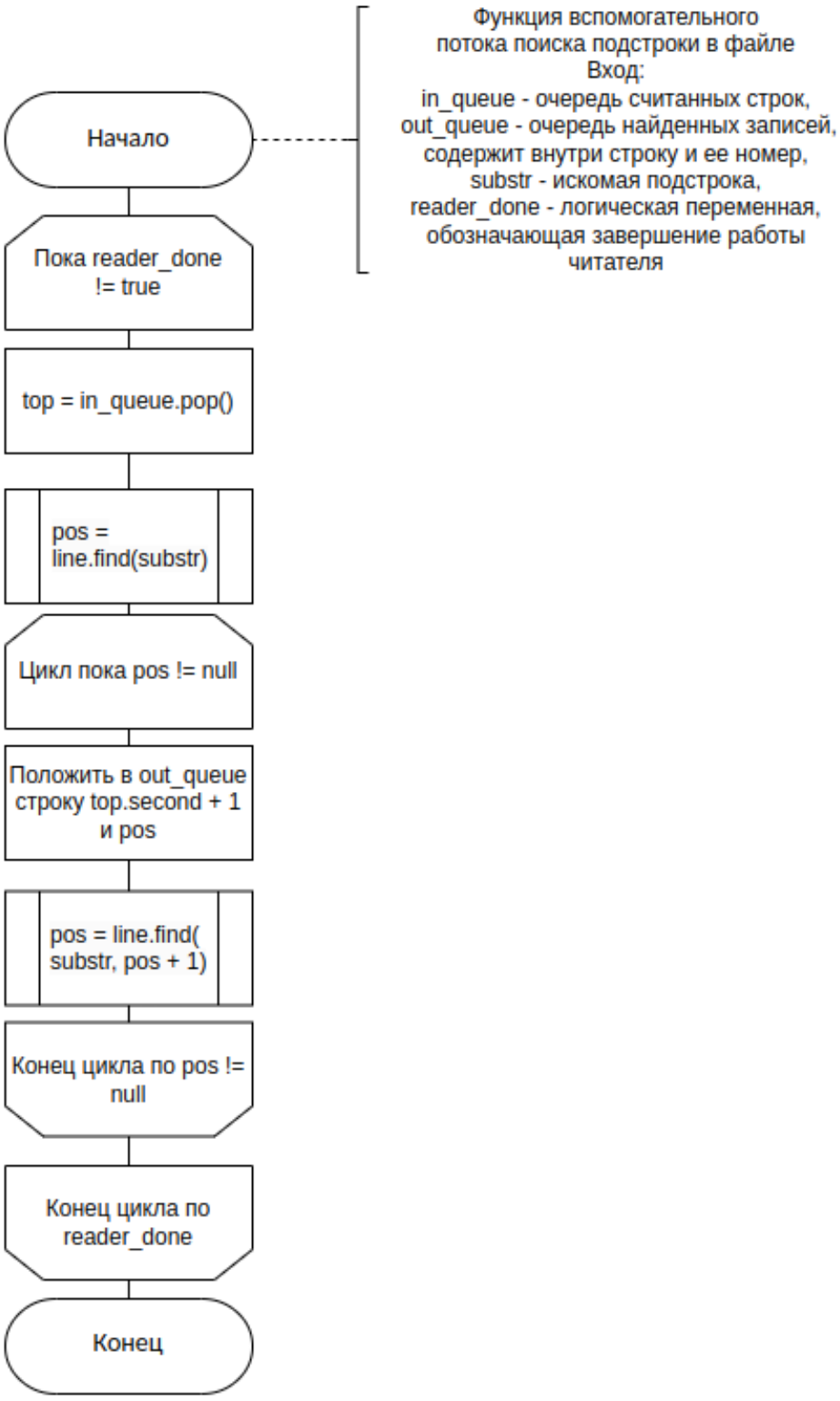

Рисунок 2.4 – Схема алгоритма работы потока, обрабатывающего считанные строки

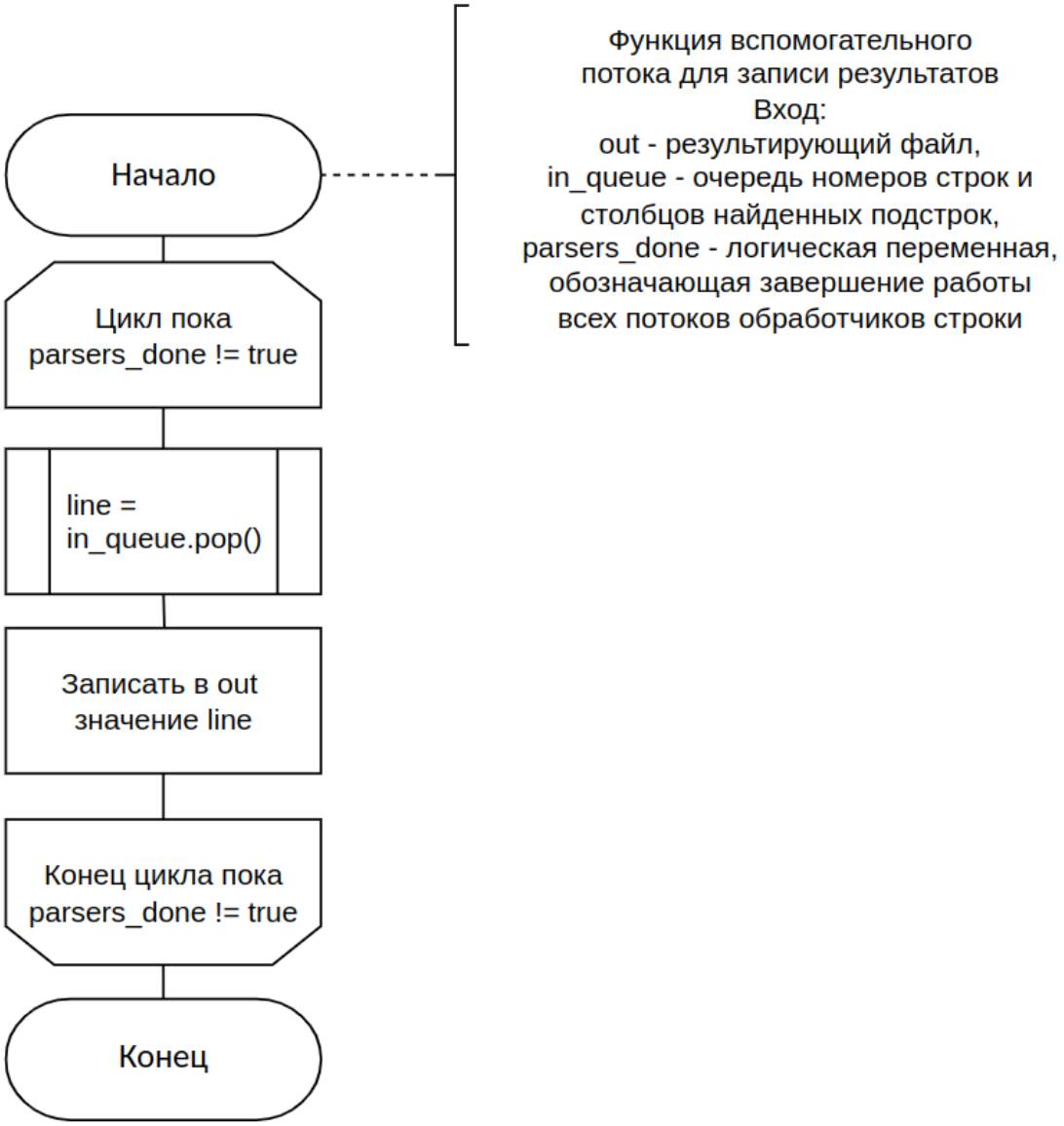

Рисунок 2.5 – Схема алгоритма работы потока, записывающего результаты в файл

Вывод

В данном разделе разработаны схемы версий рассматриваемого алгоритма

3 Технологическая часть

В данном разделе приведены средства реализации программного обеспечения и листинги кода реализаций алгоритма.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык $C++$ [3]. Данный выбор обусловлен наличием у языка встроенной библиотеки измерения количества тиков процессора, прошедших с момента запуска, и соответствием с выдвинутыми требованиям.

Для измерения времени использовалась функция *clock()* из библиотеки *ctime* [4].

3.2 Сведения о файлах программы

Данная программа разбита на следующие файлы:

— `main.cpp` — файл, содержащий точку входа в программу, в нем происходит взаимодействие с пользователем и вызов алгоритмов;

— `src/threads.cpp` — файл содержит функции, реализующие алгоритм поиска подстроки в файле;

— `src/file.cpp` — файл содержит функции для работы с файлами;

— `src/cpu_time.cpp` — файл содержит функции, замеряющие время работы написанных функций.

В листингах 3.1 – 3.5 приведены реализации алгоритма поиска подстроки в файле.

Листинг 3.1 – Функция поиска подстроки в файле с одним потоком.

```
1  int seq_substring_replace(std::string &in_file, std::string
       &out_file, std::string &substr)
2  {
3      std::ifstream in(in_file, std::ios::in);
4      if (!in.is_open())
5      {
6          return 1;
7      }
8      std::ofstream out(out_file);
9      if (!out.is_open())
10     {
11         return 1;
12     }
13
14     int total = find_file_line_total(in);
15     if (substr_search_pos(in, out, substr, 0, total))
16     {
17         in.close();
18         out.close();
19         return 1;
20     }
21
22     in.close();
23     out.close();
24     return 0;
25 }
```

Листинг 3.2 – Функция работы основного потока при многопоточной реализации

```
1  int thread_substring_replace(std::string &in_file, std::string
       &out_file, std::string &substr, size_t thread_total)
2  {
3      reader_done = false;
4      parsers_done = false;
5      std::ifstream in(in_file, std::ios::in);
6      if (!in.is_open())
7      {
8          return 1;
9      }
10     std::ofstream out(out_file);
```

```
11      if (!out.is_open())
12      {
13          return 1;
14      }
15      std::string line;
16      ThreadQueue<std::string> out_queue;
17      ThreadQueue<std::pair<std::string, int>> in_queue;
18      std::vector<std::thread> threads;
19      std::thread thread_read(read_thread, std::ref(in),
            std::ref(in_queue));
20      threads.push_back(std::move(thread_read));
21      for (int i = 0; i < thread_total; i++)
22      {
23          std::thread thread_obj(thread_substr, std::ref(substr),
                std::ref(out_queue), std::ref(in_queue));
24          threads.push_back(std::move(thread_obj));
25      }
26      std::thread thread_write(write_thread, std::ref(out),
            std::ref(out_queue));
27      for (int i = 0; i < threads.size(); i++)
28      {
29          threads[i].join();
30      }
31      {
32          std::lock_guard<std::mutex> _(parsers_done_mutex);
33          parsers_done = true;
34      }
35      thread_write.join();
36      in.close();
37      out.close();
38      return 0;
39  }
```

Листинг 3.3 – Функция работы потока-читателя

```
1  void read_thread(std::ifstream &in,
       ThreadQueue<std::pair<std::string, int>> &out_queue)
2  {
3      if (!in.is_open())
4      {
5          return;
6      }
```

```
7
8       std::string line;
9       int cur = 0;
10      while (getline(in, line))
11      {
12          out_queue.push(std::pair<std::string, int>(line, cur));
13          cur++;
14      }
15      std::unique_lock<std::shared_mutex> r(reader_done_mutex);
16      reader_done = true;
17      return;
18  }
```

Листинг 3.4 – Функция работы потока-писателя

```
1  void write_thread(std::ofstream &out, ThreadQueue<std::string>
       &in_queue)
2  {
3       if (!out.is_open())
4       {
5           return;
6       }
7
8       std::string line;
9       while (true)
10      {
11          if (in_queue.empty())
12          {
13              std::lock_guard<std::mutex> _(parsers_done_mutex);
14              if (parsers_done)
15                  return;
16              continue;
17          }
18          line = in_queue.pop();
19          out << line;
20
21      }
22      return;
23  }
```

Листинг 3.5 – Функция работы потока

```
1  void thread_substr(std::string &substr,
       ThreadQueue<std::string> &out_queue,
       ThreadQueue<std::pair<std::string, int>> &in_queue)
2  {
3      size_t pos;
4      std::pair<std::string, int> top;
5
6      while (true)
7      {
8          try
9          {
10             top = in_queue.pop();
11         }
12         catch (...)
13         {
14             std::shared_lock<std::shared_mutex>
                   r(reader_done_mutex);
15             if (reader_done)
16             {
17                 return;
18             }
19             continue;
20         }
21
22         pos = top.first.find(substr);
23         while (pos != std::string::npos)
24         {
25             std::ostringstream stream;
26             stream << top.second + 1 << " " << pos << "\n";
27             out_queue.push(stream.str());
28
29             pos = top.first.find(substr, pos + 1);
30         }
31     }
32 }
```

Вывод

В данном разделе были рассмотрены средства реализации, а также представлен листинг реализации алгоритма поиска подстроки в файле.

4 Исследовательская часть

4.1 Технические характеристики

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

— Процессор: Intel(R) Core(TM) i7-1165G7 CPU 2.80 ГГц.

— Количество ядер: 4 физических и 8 логических.

— Оперативная память: 15 Гбайт.

— Операционная система: Ubuntu 64-разрядная система версии 22.04.3.

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

4.2 Замеры времени выполнения реализаций однопоточной и многопоточной реализации

Для замеров времени использовалась функция получения значения системных часов *clock*() [5]. Функция применялась два раза — в начале и в конце измерения времени, значения полученных временных меток вычитались друг из друга для получения времени выполнения программы.

Исходный файл заполнялся случайными буквами латинского алфавита, цифрами и пробелами. Размер исходного файла — 500 МБ.

Средний размер строки — 5500 символов.

Замеры проводились по 1000 раз для количества потоков от 1 до 25 и для однопоточной программы. Значение количества потоков 0 соответствует однопоточной программе, а значения n больше 0 — программе, создающей n дополнительных потоков, обрабатывающих строки.

В таблице 4.1 представлены замеры времени выполнения программы в зависимости от количества потоков.

Таблица 4.1 – Результаты нагрузочного тестирования (в мс).

Кол-во потоков	**Время, мс**
0	203.607
1	2 459.356
2	1 298.828
3	921.261
4	726.796
5	613.520
6	447.087
7	488.265
8	502.268
9	508.111
10	511.366
11	514.586
12	517.362
13	521.815
14	524.830
15	529.721
16	532.709
17	536.419
18	538.900
19	542.836
20	546.563
21	551.605
22	554.042
23	557.653
24	562.491
25	570.482

На рисунке 4.1 представлены результаты замеров.

Рисунок 4.1 – Результаты замеров времени работы реализации алгоритма в зависимости от количества дополнительных потоков

Из полученных результатов можно сделать вывод, что однопоточный процесс работает быстрее процесса, создающего вспомогательный поток для поиска подстрок в файле. Это связано с дополнительными временными затратами на создание потока и передачи ему необходимых аргументов. Наилучший результат по времени для многопоточной реализации для показал процесс с 6 дополнительными потоками, обрабатывающими строки. Так как используются еще 2 дополнительных потока - читатель и писатель, итоговое количество дополнительных потоков равно 8, что соответствует количеству логических ядер устройства. Для числа потоков, большего данной величины, затраты на содержание потоков превышают преимущество от использования многопоточности, и функция времени от количества потоков начинает расти.

Временные затраты однопоточной реализации были меньше многопоточной при всех рассматриваемых количествах потоков, что объясняется затратами времени на передачу данных между потоками.

4.3 Вывод

В результате замеров было выявлено, что увеличение количества потоков может увеличить производительность для многопоточной реализации, однако затраты времени на поддержку множества потоков и передачу данных между ними приводит к увеличению временных затрат по сравнению с однопоточной реализацией.

Выборка из результатов замеров времени (для 5632 документов):

— однопоточный процесс — 203 мс;

— один дополнительный поток, выполняющий все вычисления — 2459 мкс;

— 6 потоков (лучший результат) — 447 мкс, что в 5.5 раз быстрее выполнения процесса с одним потоком;

— 25 потока (худший результат) — 570 мкс, что в 4.3 раза быстрее выполнения процесса с одним потоком.

Таким образом, однопоточная реализация затрачивает меньше времени при любом количества дополнительных потоков многопоточной программы. Добавление дополнительных потоков может как ускорить выполнение многопоточной программы, так и замедлить. Рекомендуется использовать на данной архитектуре ЭВМ число дополнительных потоков, равное числу логических ядер устройства.

Заключение

В ходе выполнения лабораторной работы было выявлено, что многопоточная реализация алгоритма поиска подстроки медленнее, чем однопоточная. Увеличение количества потоков может как уменьшить время выполнения программы, так и увеличить.

Цель, поставленная в начале работы, была достигнута: были получены навыки организации параллельного выполнения операций. Были достигнуты все поставленные задачи.

1) Описаны основы распараллеливания вычислений.

2) Разработано программное обеспечение, которое реализует однопоточный алгоритм нахождения всех вхождений подстроки в файле.

3) Разработана и реализована многопоточная версия данного алгоритма.

4) Определены средства программной реализации.

5) Выполнены замеры реального времени работы реализаций алгоритма.

6) Проведено сравнение по времени работы реализаций алгоритма.

При значении потоков превышающем число логических ядер (более 8 для устройства, на котором проводилось тестирование), затраты на содержание потоков превышают преимущество от использования многопоточности и время выполнения по сравнению с лучшим результатом (для 8 потоков) растут.

Список использованных источников

1. Stoltzfus Justin. Multithreading. — Режим доступа: https://www.techopedia.com/definition/24297/multithreading-computer-architecture (дата обращения: 28.01.2023).

2. У.Р. Стивенс, С.А. Раго. UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018. — С. 994.

3. Документация по Microsoft C++ [Электронный ресурс]. — Режим доступа: https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017 (дата обращения: 25.09.2023).

4. C library function clock() [Электронный ресурс]. — Режим доступа: https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm (дата обращения: 25.09.2023).

5. Определение текущего времени с высокой точностью [Электронный ресурс]. — Режим доступа: http://all-ht.ru/inf/prog/c/func/clock_gettime.html (дата обращения: 28.01.2023).