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»

Тема: «Конвейерная обработка данных»

Студент: Пронин А. С.

Группа: ИУ7-52Б

Преподаватель: Волкова Л. Л.

Оценка: _____

Москва

2021

Содержание

Введение

Цель работы – получить навык организации асинхронной передачи данных между потоками на примере конвейерной обработки информации.

Задачи работы:

- выбрать и описать методы обработки данных, которые будут сопоставлены методам конвейера;
- описать архитектуру программы, а именно какие функции имеет главный поток, принципы и алгоритмы обмена данными между потоками;
- реализовать конвейерную систему, а также сформировать лог событий с указанием времени их происхождения, описать реализацию;
- интерпретировать сформированный лог;

1 Аналитический раздел

Выполнение каждой команды складывается из ряда последовательных этапов (шагов, стадий), суть которых не меняется от команды к команде. С целью увеличения быстродействия процессора и максимального использования всех его возможностей в современных микропроцессорах используется конвейерный принцип обработки информации. Этот принцип подразумевает, что в каждый момент времени процессор работает над различными стадиями выполнения нескольких команд, причем на выполнение каждой стадии выделяются отдельные аппаратные ресурсы. По очередному тактовому импульсу каждая команда в конвейере продвигается на следующую стадию обработки, выполненная команда покидает конвейер, а новая поступает в него.

Конвейерная обработка в общем случае основана на разделении подлежащей исполнению функции на более мелкие части 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.

1.1 Алгоритм

В данной лабораторной работе выбран следующий алгоритм для реализации: поиск наибольшего полинома в строке. Данный алгоритм можно разбить на 3 этапа:

- разбить строку на слова;
- найти все полиномы в полученных словах;
- найти наибольший полином из всех;

1.2 Вывод

В данном разделе была расмотрена концепция конвейера и выбран алгоритм для реализации.

2 Конструкторский раздел

В данном разделе представлены схемы алгоритмов главного потока и конвейера.

2.1 Схемы алгоритмов

Схема алгоритма главного потока:

Схема алгоритма конвейера:

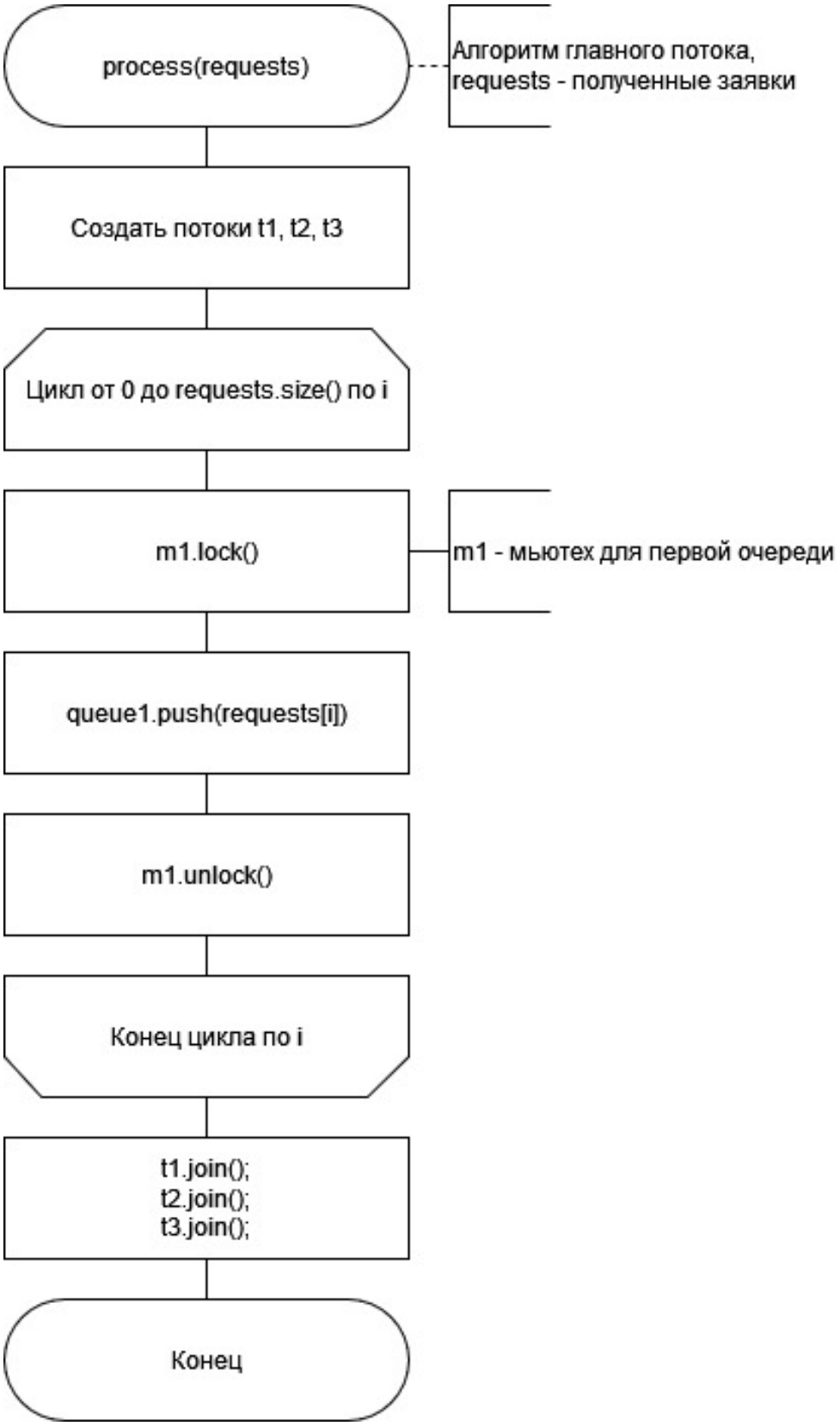

Рис. 2.1: Схема алгоритма главного потока

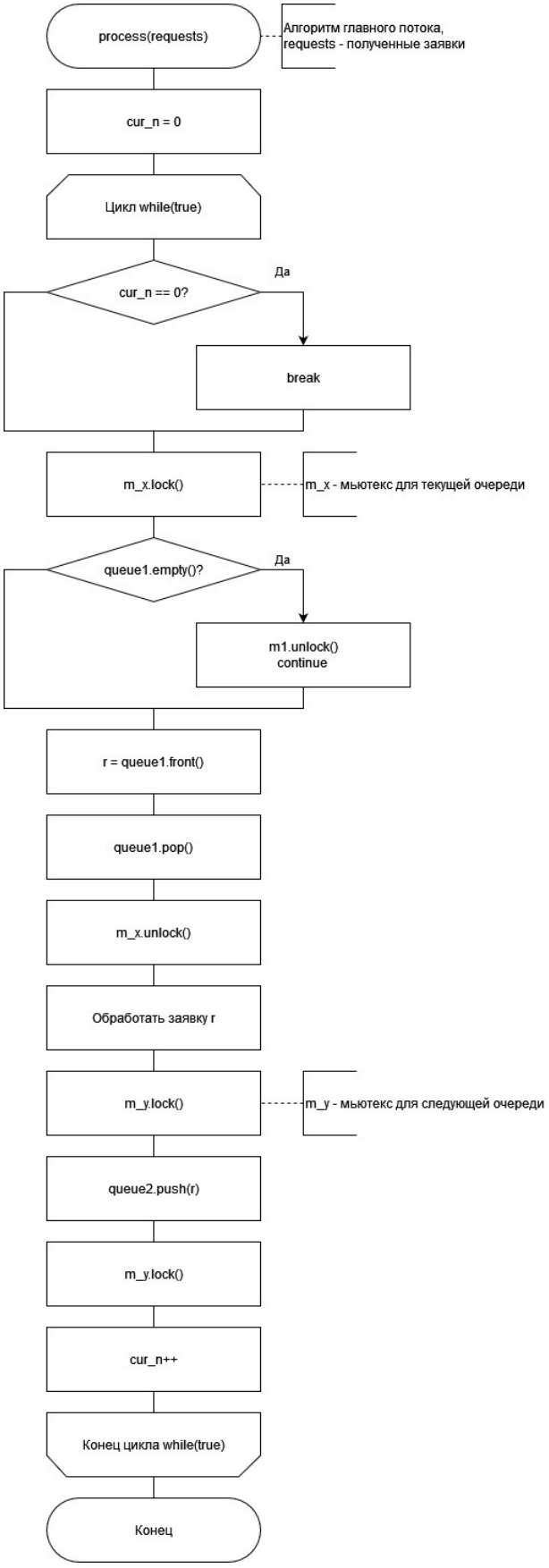

Рис. 2.2: Схема алгоритма конвейера

2.2 Вывод

В данном разделе были разработаны схемы главного потока и конвейера.

3 Технологический раздел

В данном разеделе представлены требование к ПО, выбор инструментов для реализации и оценки алгоритмов, а также листинги полученного кода.

3.1 Требование к ПО

К программе предъявляется ряд требований:

- на вход подаётся строка;
- на выходе — самый длинный полином.

3.2 Выбор инструментов

По-скольку наиболее освоенным языком для разработчика является c++, для реализации поставленной задачи был выбран именно он, т.к. таким образом работа будет проделана наиболее быстро и качественно.

Соответсвенно для компиляции кода будет использоваться компилятор G++.

Чтобы оценить время выполнения программы будет замерятся реальное время, т.к. таким образом можно будет сравнить реализацию с конвеером и без. Для замера реального времени работы программы используется функция $chrono :: high\_resolution\_clock :: now()$ т.к. программа тестируется на компьютере с установленной ОС Windows. [1]

Кроме этого, необходимо отключить оптимизации компилятора для более честного сравнения алгоритмов. В моём случае это делается с помощью ключа $-O0$ т.к. используется компилятор G++. [2]

3.3 Реализация алгоритмов

На листингах 3.1-3.4 представлены реализации алгоритма ZBuffer без и с использованием параллельных вычислений.

Листинг 3.1: Алгоритм главного потока

```
void Conveyor::process(vector<Request> &requests)
{
    start_time = Clock::now();

    thread t1(&Conveyor::firstBent, this);
    thread t2(&Conveyor::secondBent, this);
    thread t3(&Conveyor::thirdBent, this);

    for (size_t i = 0; i < requests.size(); i++)
    {
        m1.lock();
        requests[i].push_time1 = duration_cast<nanoseconds>(Clock::now() -
            start_time).count()/1000000000.;
        queue1.push(requests[i]);
        m1.unlock();
    }

    t1.join();
    t2.join();
    t3.join();
}
```

Листинг 3.2: Алгоритм первой ленты конвеера

```
void Conveyor::firstBent()
{
    int cur_n = 0;
    while (true)
    {
        if (cur_n == n)
            break;
        m1.lock();
        if (queue1.empty())
        {
            m1.unlock();
            continue;
        }
        Request r = queue1.front();
        queue1.pop();
        r.pop_time1 = duration_cast<nanoseconds>(Clock::now() -
            start_time).count()/1000000000.;
        m1.unlock();
        r.getWords();
        m2.lock();
        r.push_time2 = duration_cast<nanoseconds>(Clock::now() -
            start_time).count()/1000000000.;
        queue2.push(r);
        m2.unlock();
        cur_n ++;
    }
}
```

Листинг 3.3: Алгоритм второй ленты конвеера

```
void Conveyor::secondBent()
{
    int cur_n = 0;
    while (true)
    {
        if (cur_n == n)
            break;
        m2.lock();
        if (queue2.empty())
        {
            m2.unlock();
            continue;
        }
        Request r = queue2.front();
        queue2.pop();
        r.pop_time2 = duration_cast<nanoseconds>(Clock::now() -
            start_time).count()/1000000000.;
        m2.unlock();
        r.getPolinoms();
        m3.lock();
        r.push_time3 = duration_cast<nanoseconds>(Clock::now() -
            start_time).count()/1000000000.;
        queue3.push(r);
        m3.unlock();
        cur_n ++;
    }
}
```

Листинг 3.4: Алгоритм третей ленты конвеера

```
void Conveyor::thirdBent()
{
    int cur_n = 0;
    while (true)
    {
        if (cur_n == n)
            break;
        m3.lock();
        if (queue3.empty())
        {
            m3.unlock();
            continue;
        }
        Request r = queue3.front();
        queue3.pop();
        r.pop_time3 = duration_cast<nanoseconds>(Clock::now() -
            start_time).count()/1000000000.;
        m3.unlock();
        r.getLongestPolinom();
        r.processing_time = duration_cast<nanoseconds>(Clock::now() -
            start_time).count()/1000000000.;

        m4.lock();
        res.push_back(r);
        m4.unlock();
        cur_n ++;

    }
}
```

3.4 Тестирование

Для проверки написанных алгоритмов были подготовлены следующие тесты:

- проверка реузльтата обработки строки "test lol 23323232 s sss ll 2332323322 sls sll lls"
- проверка реузльтата обработки строки "test lol 23323232 s sss ll 233232332 sls sll lls"
- проверка реузльтата обработки строки "te str"

Для подготовленных тестов ожидаются следующие результаты соответсвтенно:

- "lol"
- "233232332"
- "No polinoms"

На рисунке 3.1 приведены результаты тестирования.

```
Request #1:
str = test lol 23323232 s sss ll 2332323322 sls sll lls
words = test; lol; 23323232; s; sss; ll; 2332323322; sls; sll; lls;
polinoms = lol; s; sss; ll; sls;
longest_polinom = lol

Request #2:
str = test lol 23323232 s sss ll 233232332 sls sll lls
words = test; lol; 23323232; s; sss; ll; 233232332; sls; sll; lls;
polinoms = lol; s; sss; ll; 233232332; sls;
longest_polinom = 233232332

Request #3:
str = te str
words = te; str;
polinoms =
longest_polinom = No polinoms
```

Рис. 3.1: Результаты тестирования

Как видно по рисунку, все тесты пройдены.

3.5 Вывод

В данном разделе были выдвинуты требования к ПО, выбраны инструменты для реализации выбранных алгоритмов, представлены листинги реализованных алгоритмов, а также проведено тестирование.

4 Исследовательский раздел

В данном разделе представлены технические характеристики компьютера, используемого для тестирования, и сравнительный анализ реализованных алгоритмов.

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- операционная система: Windows 10 (64-разрядная);
- оперативная память: 32 GB;
- процессор: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz;
- количество ядер: 4;
- количество потоков: 8.

4.2 Сравнительный анализ времени выполнения алгоритмов

Чтобы провести сравнительный анализ времени выполнения алгоритмов, замерялось реальное время работы алгоритма ZBuffer 100 раз и делилось на кол-во итераций. В таблице 4.1 показаны результаты тестирования алгоримта, использующего параллельные вычисления, для разного кол-ва потоков.

Таблица 4.1: Таблица времени выполнения алгоритма ZBuffer с использованием параллельных вычислений, при размере карты высот 33x33

Количество потоков, шт	Паралльная реализация алгоритма, сек
1	0.12207
2	0.10223
4	0.08782
8	0.08062
16	0.08563
32	0.09113

Из таблицы выше видно, что наибольший выигрышь по времени даёт алгоритм использующий 8 потоков, поэтому для сравнения с обычным алгоритмом будем использовать именно его. В таблице 4.2 приведены результаты этого сравнения.

Таблица 4.2: Таблица времени выполнения обычного ZBuffer и с использованием параллельных вычислений на 8 потоках ZBuffer

Размер карты высот, шт	Обычный, сек	Параллельный (8 потоков), сек
33x33	0.12056	0.08062
65x65	0.12524	0.08262
129x129	0.15733	0.08707
257x257	0.21563	0.09875
513x513	0.35673	0.12993

4.3 Вывод

По итогу иследования выяснилось, что разработанные алгоритмы работают верно, то-есть удаляют невидимые линии и поверхности. Кроме этого, смотря на результаты сравнительного анализа времени выполнения обычного и многопоточного алгоритмов ZBuffer, логично сделать вывод, что наиболее быстрым, является алгоритм, использующий параллельные вычисления.

Заключение

По итогу проделанной работы была достигнута цель - изучены параллельные вычисления с помощью потоков на примере алгоритма ZBuffer.

Также были решены все поставленные задачи, а именно:

- изучены распараллеливание вычислений и работа с потоками;
- реализованы стандартный алгоритм ZBuffer, а также с применением параллеленных вычислений;
- проведён сравнительный анализ времени работы алгоритма при разном количестве потоков.

Список использованных источников

[1] <chrono>. // URL: https://docs.microsoft.com/ru-ru/cpp/standard-library/chrono?view=msvc-160&viewFallbackFrom=vs-2017.

[2] Как применить настройки оптимизации gcc в qt? // URL: http://blog.kislenko.net/show.php?id=1991.