Отчет по лабораторной работе №7
по курсу "Анализ алгоритмов"
по теме "Вычислительный конвейер"

Студент: Доктор А.А. ИУ7-53
Преподаватель: Волкова Л.Л., Строганов Ю.В.

2018 г.

Содержание

Введение

Конвейер — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация параллелизма на уровне инструкций), технология, используемая при разработке компьютеров и других цифровых электронных устройств.

Описание

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.

Идея заключается в параллельном выполнении нескольких инструкций процессора. Сложные инструкции процессора представляются в виде последовательности более простых стадий. Вместо выполнения инструкций последовательно (ожидания завершения конца одной инструкции и перехода к следующей), следующая инструкция может выполняться через несколько стадий выполнения первой инструкции. Это позволяет управляющим цепям процессора получать инструкции со скоростью самой медленной стадии обработки, однако при этом намного быстрее, чем при выполнении эксклюзивной полной обработки каждой инструкции от начала до конца [1].

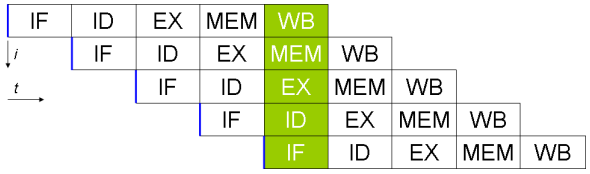

Рис. 1: Простой пятиуровневый конвейер в RISC-процессорах

На иллюстрации 1 показан простой пятиуровневый конвейер в RISC-процессорах. Здесь:

1) IF (англ. Instruction Fetch) — получение инструкции;
2) ID (англ. Instruction Decode) — раскодирование инструкции;
3) EX (англ. Execute) — выполнение;
4) MEM (англ. Memory access) — доступ к памяти;
5) WB (англ. Register write back) — запись в регистр.

Вертикальная ось — последовательные независимые инструкции, горизонтальная — время. Зелёная колонка описывает состояние процессора в один момент времени, в ней самая ранняя, верхняя инструкция уже находится в состоянии записи в регистр, а самая последняя, нижняя инструкция — только в процессе чтения.

Применение

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 обработкии нформации. 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 [2].

Виды

Перечислим основные типы конвейеров:

1) Арифметический конвейер (arithmetic pipeline) — реализация в АЛУ поэтапного исполнения арифметических операций чаще всего над вещественными числами [3];

2) супер-конвейер, гипер-конвейер, глубокий конвейер (super-pipeline, hyper-pipeline, deep pipeline) — вычислительный конвейер с необычно большим количеством стадий. Например, процессор Intel Pentium 4 имел 20 стадий конвейера, а в модификации Prescott получил конвейер из 31 стадии [4];

3) недозагруженный конвейер (underutilized pipeline) — конвейер, в котором в одно и то же время не все стадии конвейера выполняют какую-то операцию. Например ранние процессоры MIPS имели 6-стадийный конвейер, но в каждый момент было занято только 3 его стадии.

Задачи работы

В ходе выполнения данной лабораторной работы, мной были выполнены следующие задачи:

1) Построить вычислительный конвейер;

2) провести замеры скорости работы работы конвейера при разной нагруженности блоков.

1 Аналитическая часть

В данном разделе приведена оценка производительности идеального конвейера.

1.1 Оценка производительности идеального конвейера

Пусть задана операция, выполнение которой разбито на n последовательных этапов. При последовательном их выполнении операция выполняется за время

$$\tau_e = \sum_{i=1}^{n} \tau_i \tag{1}$$

где

n – количество последовательных этапов;

τ_i – время выполнения i-го этапа;

Быстродействие одного процессора, выполняющего только эту операцию, составит

$$S_e = \frac{1}{\tau_e} = \frac{1}{\sum\limits_{i=1}^{n} \tau_i} \tag{2}$$

где

τ_e – время выполнения одной операции;

n – количество последовательных этапов;

τ_i – время выполнения i-го этапа;

Выберем время такта — величину $t_T = max\sum\limits_{i=1}^{n}(\tau_i)$ и потребуем при разбиении на этапы, чтобы для любого $i = 1, \ldots, n$ выполнялось условие $(\tau_i + \tau_{i+1}) \bmod n = \tau_T$. То есть чтобы никакие два последовательных этапа (включая конец и новое начало операции) не могли быть выполнены за время одного такта.

Максимальное быстродействие процессора при полной загрузке конвейера составляет

$$S_{max} = \frac{1}{\tau_T} \tag{3}$$

где

τ_T – выбранное нами время такта;

Число n — количество уровней конвейера, или глубина перекрытия, так как каждый такт на конвейере параллельно выполняются n операций. Чем

больше число уровней (станций), тем больший выигрыш в быстродействии может быть получен.

Известна оценка

$$\frac{n}{n/2} \leq \frac{S_{max}}{S_e} \leq n \tag{4}$$

где

S_{max} – максимальное быстродействие процессора при полной загрузке конвейера;

S_e – стандартное быстродействие процессора;

n – количество этапов.

то есть выигрыш в быстродействии получается от n/2 до n раз [5].

Реальный выигрыш в быстродействии оказывается всегда меньше, чем указанный выше, поскольку:

1) некоторые операции, например, над целыми, могут выполняться за меньшее количество этапов, чем другие арифметические операции. Тогда отдельные станции конвейера будут простаивать;

2) при выполнении некоторых операций на определённых этапах могут требоваться результаты более поздних, ещё не выполненных этапов предыдущих операций. Приходится приостанавливать конвейер;

3) поток команд(первая ступень) порождает недостаточное количество операций для полной загрузки конвейера.

2 Конструкторская часть

В данном разделе будут описаны принципы работы выбранных решений и их блоксхемы.

2.1 Разработка реализаций алгоритмов

Моя программа "собирает"танки из трёх компонентов: корпуса, двигателя и пушки. Итого 3 этапа сборки. Один этап представлена классом TankQueue, который хранит следующие поля:

1) tanks - Очередь танков, ожидающих выполнения действия;
2) work - функтор - функция постройки;
3) queue_mutex - временной(для установки времени ожидания) мьютекс для блокировки доступа к очереди выполнения;
4) main - флаг первой активной очереди;
5) build_time - время постройки;
6) downtime - время ожидания появления элементов в очереди;
7) time_for_mutex - время ожидания на мьютексе;
8) id - идентификатор танка для отладки.

Я упомянул очередь танков. Под танками я подразумеваю класс Tank с разными методами сборки(в них нет ничего интересного, они будут приведены ниже в соответсвующем листинге). Единственное важное, что у танка тоже есть свой мьютекс - tank_mutex.

Итак, как работает мой конвейер. Создаются 5 очередей TankQueue. В конвейере 3 составляющих, а очередей 5, почему так? 2 очереди отведены просто под хранилища танков: первоначальные танки(пустые) и конечные(полностью построенные). Оставшиеся три предназначены для установки корпуса, двигателя и пушки. Всем очередям присваивается флаг конца работы False. Очередь пустых танков является первой очередью, поэтому ее флаг main устанавливается в True. В очередь готовых к исполнению танков для класса TankQueue пустых танков заносится очередь из 5 танков. Всё, на этом приготовления закончены. Все составляющие конвейера готовы к работе. Танк считается построенным, если он получил корпус, двигатель и пушку - см. рис. 2

Рис. 2: Алгоритм построения танка

Запускаем потоки для каждой очереди. Функцией потока выбирается метод start, принимающий на вход ссылку на следующую очередь. Принцип ее работы изображен на рис. 4 и рис. 5. Функция постоянно ожидает, когда флаг конца работы(finish) станет True. Если же он False, то функция смотрит, хранится ли в очереди ее класса какие нибудь танки, которые надо обработать. Если да, то захватываем мьютекс очереди выполнения, достаём первый танк и освобождаем мьютекс. Далее выполняем работу над танком. Помните функтор work? Вот его и вызываем. Ему на вход требуется танк и время выполнения. Танк есть, а время выполнения хранится в buildtime. Его принцип работы изображен на рис. 3.

Рис. 3: Установка компонента в танк

После того как работа выполнена пытаемся подключить к очереди следующего этапа конвейера. Ожидаем на мьютексе следующей очереди time_for_mutex секунд. Если подключиться не удается засыпаем на некоторое время, чтобы подождать освобождения мьютекса. Этот цикл закончится только после того, как мы достигнем нашей цели и захватим мьютекс. Как только мы его захватываем, мы кладем в конец очереди обработанный нами танк и освобождаем мьютекс. Так мы крутимся в нашей очереди выполнения, пока она не опустеет. Как только она опустеет, выходим из нее и смотрим, является ли данный класс главный этапом конвейера. Если да, то еще раз убеждаемся, что очередь пустая. Под главным этапом я подразумеваю такой этап, в который нельзя послать новые элементы, поскольку он является первым. Итак, если текущий этап таковой и его очередь пуста, то назначаем следующий этап главным, а у себя ставим флаг finish в True - то есть прекращаем работу. Если же текущий этап не являлся не главным, то засыпаем на время downtime, ожидая, пока в нашу очередь добавят новых танков. После чего просыпаемся и по новой. Сигнал завершения был? Нет? Тогда пока очередь не пуста, обрабатываем нашу очередь, занося обработанные танки в следующую и надеемся, что нашу очередь сделали главной и мы можем прекратить работу и пойти отдыхать, назначив главной следующую. А теперь тоже самое, но в виде схемы - рисунки 4, 5.

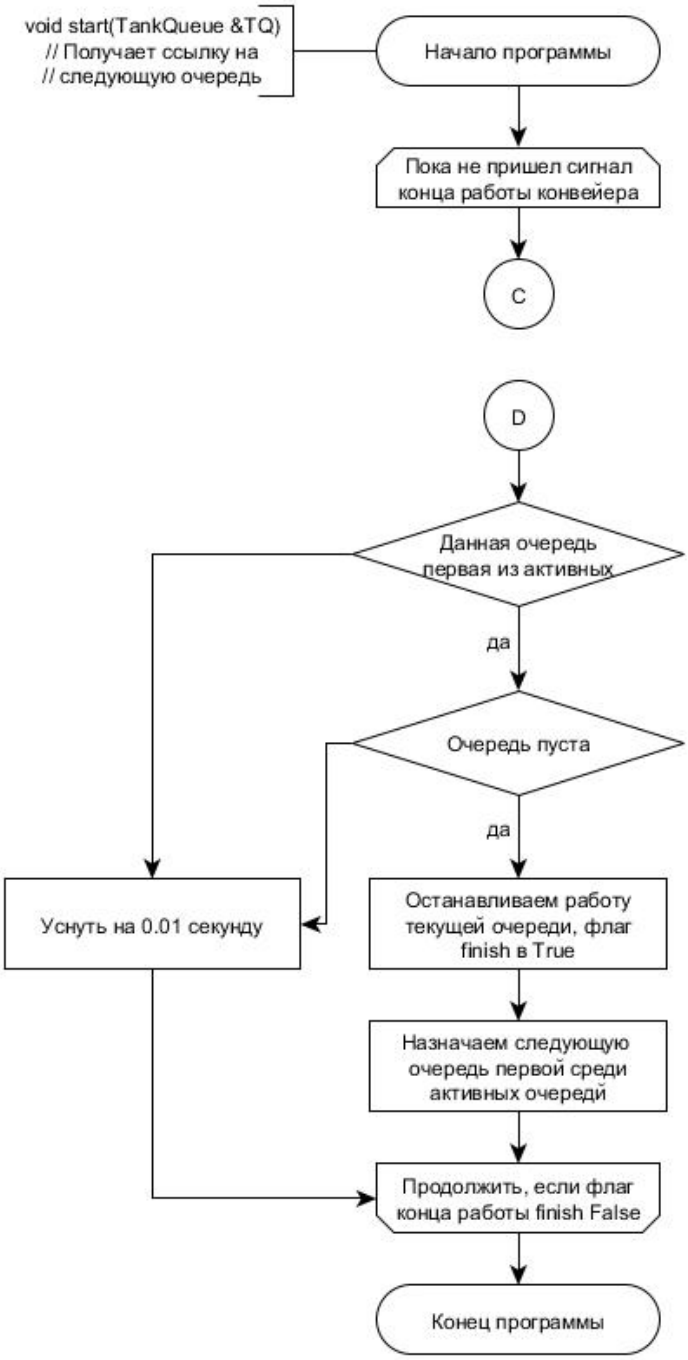

Рис. 4: Цикл работы, пока не наступил флаг конца работы.

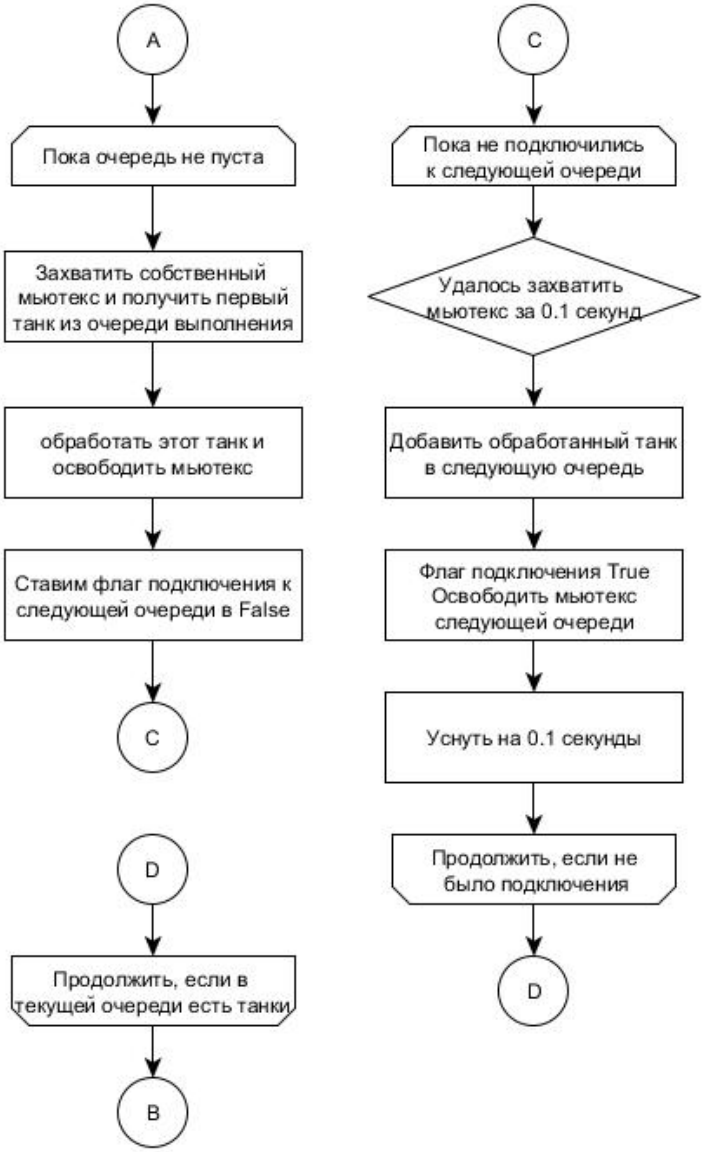

Рис. 5: Обработка танка и добавление его в следующую очередь.

3 Технологическая часть

В данном разделе будут определены средства реализации и приведен листинг кода.

3.1 Средства реализации

Не изменяю себе и вновь использую C++. Среда разработки - Qt. Время работы процессора замерялось с помощью функции, продемонстри-

рованной в листинге 1. Эта функция в отличие от встроенной функции таймера, способна считать реальное процессорное время работы программы в тиках [7]. Для ее работы была подключена библиотека time.h. Для работы с потоками и мьютексами применялись библиотеки windows.h(Для сна Sleep), mutex, thread. Контейнером очереди был std::queue, поэтому также использовалась библиотека queue.

Листинг 1: Функция замера процессороного времени

```
1 unsigned long long tick(void)
2 {
3    unsigned long long d;
4    __asm__ __volatile__ ("rdtsc" : "=A"     (d));
5    return d;
6 }
7
```

3.2 Реализация алгоритмов

Как я упомянал в конструкторской части, были написаны классы Tank(листинг 2) и TankQueue(листинг 3).

Листинг 2: Класс Tank

```
1   class Tank : public Unit
2   {
3   public:
4      static int count;
5      int id;
6      std::mutex tank_mutex;
7
8      Tank(){id = Tank::count++; built = false;}
9       ~Tank(){}
10
11     void insert_weapon(Weapon weapon, DWORD time) {
12        Sleep(time);
13        _weapon = weapon;
14     }
15
16     void insert_body(Body body, DWORD time) {
17        Sleep(time);
18        _body = body;
19     }
20
21     void insert_engine(Engine engine, DWORD time) {
22        Sleep(time);
23        _engine = engine;
24     }
25
26     bool has_engine(){return _engine.is_built();}
27     bool has_weapon(){return _weapon.is_built();}
28     bool has_body(){return _body.is_built();}
29
30     bool is_ready() {return _engine.is_built() && _body.is_built() &&
    _weapon.is_built();}
31
32     bool is_built() {return built;}
33
34
35     void complete() {
36        if (_engine.is_built() && _body.is_built() && _weapon.is_built()) {
37           Sleep(50);
38           built = true;
39        }
40     }
41
42     // Конструктор копирования для queue.pop
43     Tank(const Tank & other) {
44        _engine = other._engine;
45        _body = other._body;
46        _weapon = other._weapon;
47        id = other.id;
```

```
48          }
49
50          Tank & operator=(const Tank & other) {
51              _engine = other._engine;
52              _body = other._body;
53              _weapon = other._weapon;
54              id = other.id;
55
56              return *this;
57          }
58
59          void move(int speed);
60          void rotate(double angle);
61          void fire();
62
63      private:
64          bool built;
65          Engine _engine;
66          Body _body;
67          Weapon _weapon;
68      };
69
```

Листинг 3: Класс TankQueue

```
1       class TankQueue {
2   private:
3       // Очередь готовых на выполнение
4       std::queue<Tank> tanks;
5       // Функтор - функция постройки
6       std::function<Tank(Tank &tank, DWORD time)> work;
7       // Мьютекс для блокировки доступа к очереди выполнения
8       std::timed_mutex queue_mutex;
9       // флаг конца работы, можно было сделать без него, но так понятнее
10      bool finish;
11      // флаг первого рабочего конвейера
12      bool main;
13  public:
14      // Время ожидания на мьютексе
15      DWORD time_for_mutex;
16      // Время ожидания появления элементов в очереди
17      DWORD downtime;
18      // время постройки - занесено сюда для тестирования
19      DWORD build_time;
20      // идентификатор танка - занесен сюда для отладки
21      int id;
22      // статическая переменаня нужна для id
23      static int count;
24
25      // Конструкторы копирования и перемещения
26      TankQueue(const TankQueue & other);
27
28      TankQueue(const TankQueue && other);
29
```

```
30      // Для очереди функций работы с танками
31      TankQueue(std::function<Tank(Tank&, DWORD)> f, DWORD t);
32
33      // Для очереди готовых танков
34      TankQueue(){} // не использовать больше нигде
35
36      // Для очереди пустых танков
37      TankQueue(size_t amount);
38
39      void set_main() {
40          main = true;
41      }
42
43      void start(TankQueue &TQ);
44
45      Tank pop() {
46          Tank tank = tanks.front();
47          tanks.pop();
48          return tank;
49      }
50
51      void push(Tank &tank) {
52          tanks.push(tank);
53      }
54
55      size_t size() {
56          return tanks.size();
57      }
58  };
59
```

Кода как всегда много, поэтому для упрощения маршрутизации по отчету, приведу здесь гиперссылки по оставшимся листингам. Нумерация от наиболее важных функций к второстепенным:

листинг 15 - класс Weapon;

листинг 16 - дополнительные классы и определение статических переменных.

Листинг 4: Функция start.

```
void TankQueue::start(TankQueue &TQ){
    finish_mutex.lock();

    while (!finish) {
        finish_mutex.unlock();
        queue_mutex.lock();
        while (tanks.size() > 0) {
            times_in.push_back(clock());
            Tank tank = pop();
            queue_mutex.unlock();
            tank = work(tank, build_time);
            bool connect = 0;
            while (!connect) {
                if (TQ.queue_mutex.try_lock_for(
                        Ms(time_for_mutex))) {
                    TQ.push(tank);
                    times_out.push_back(clock());
                    connect = 1;
                    TQ.queue_mutex.unlock();
                }
                std::this_thread::sleep_for(Ms(50));
            }
            queue_mutex.lock();
        }
        queue_mutex.unlock();
        main_mutex.lock();
        if (main && tanks.size() == 0) {
            main_mutex.unlock();

            finish_mutex.lock();
            finish = 1;
            finish_mutex.unlock();

            TQ.main_mutex.lock();
            TQ.set_main();
            TQ.main_mutex.unlock();


            end = clock();
        } else {
            main_mutex.unlock();
            std::this_thread::sleep_for(Ms(downtime));
        }
        finish_mutex.lock();
    }
    //std::cout << "\nfinish";
    finish_mutex.unlock();
}
```

```
49
```

Листинг 5: Функция add_engine - встроить двигатель.

```
1  Tank add_engine(Tank &tank, DWORD time) {
2      tank.tank_mutex.lock();
3
4      if (!tank.has_engine()) {
5          Engine engine;
6          tank.insert_engine(engine, time);
7          std::cout << "\n add engine to tank " << tank.id;
8      }
9
10     tank.tank_mutex.unlock();
11
12     return tank;
13 }
14
```

Листинг 6: Функция add_weapon - встроить оружие.

```
1  Tank add_weapon(Tank &tank, DWORD time) {
2      tank.tank_mutex.lock();
3
4      if (!tank.has_weapon()) {
5          Weapon weapon;
6          tank.insert_weapon(weapon, time);
7          std::cout << "\n add weapon to tank " << tank.id;
8      }
9
10     tank.tank_mutex.unlock();
11
12     return tank;
13 }
14
```

Листинг 7: Функция add_body - встроить корпус.

```
1    Tank add_body(Tank &tank, DWORD time) {
2      tank.tank_mutex.lock();
3
4      if (!tank.has_body()) {
5          Body body;
6          tank.insert_body(body, time);
7          std::cout << "\n add body to tank " << tank.id;
8      }
9      tank.tank_mutex.unlock();
10     return tank;
11 }
12
```

Листинг 8: Функция main.

```
int main()
 {
    Tank::count = 1;
    TankQueue::count = 1;

    size_t amount = 5;
    TankQueue tanks(amount);

    TankQueue queue_engine(add_engine, 90);
    TankQueue queue_body(add_body, 30);
    TankQueue queue_weapon(add_weapon, 200);

    TankQueue ready_tanks;

    std::thread give_to_body([&tanks, &queue_body](){ tanks.start(queue_body)
   ;});
    std::thread give_to_engine([&queue_body,
    &queue_engine](){
        queue_body.start(queue_engine);
        });

    std::thread give_to_weapon([&queue_engine,
    &queue_weapon](){
        queue_engine.start(queue_weapon);
        });

    std::thread give_to_ready([&queue_weapon,
    &ready_tanks](){
        queue_weapon.start(ready_tanks);
        });

    give_to_body.join();
    give_to_engine.join();
    give_to_weapon.join();
    give_to_ready.join();

    std::cout << "\n\ni finished! " << ready_tanks.size() << " " <<
   queue_weapon.size() <<  queue_engine.size() << queue_body.size();

    return 0;
 }

```

Листинг 9: Функция TankQueue(size_t amount) - конструктор класса для пустой очереди.

```
TankQueue::TankQueue(size_t amount){
    id = TankQueue::count++;
    for (size_t i = 0; i < amount; i++) {
        Tank tank;
        tanks.push(tank);
    }
    main = 1;
```

```
8       finish = 0;
9       work = noWork;
10      time_for_mutex = 100;
11      downtime = 10;
12  }
13
```

Листинг 10: Конструктор класса для очереди построения компонента.

```
1     TankQueue::TankQueue(std::function<Tank(Tank&, DWORD)> f, DWORD t){
2       id = TankQueue::count++;
3       main = 0;
4       finish = 0;
5       work = f;
6       build_time = t;
7       time_for_mutex = 100;
8       downtime = 10;
9   }
10
```

Листинг 11: Конструктор копирования.

```
1   TankQueue::TankQueue(const TankQueue & other) {
2       id = other.id;
3       tanks = other.tanks;
4       work = other.work;
5       finish = other.finish;
6       main = other.main;
7
8       time_for_mutex = other.time_for_mutex;
9       downtime = other.downtime;
10  }
11
```

Листинг 12: Конструктор перемещения.

```
1   TankQueue::TankQueue(const TankQueue && other) {
2       id = other.id;
3
4       tanks = other.tanks;
5       work = other.work;
6
7       finish = other.finish;
8       main = other.main;
9
10      time_for_mutex = other.time_for_mutex;
11      downtime = other.downtime;
12  }
13
```

Листинг 13: Вспомогательный класс Body.

```
class Body : public Component
{
public:
  Body(){built = false;}

  void build() {Body::count++; built = true;}

  bool is_built(){return built;}

  static int count;

protected:
   bool built;

   int max_health;
   int health;
   int rotation_speed;
   int current_rotation_angle;
};

```

Листинг 14: Вспомогательный класс Engine.

```
class Engine : public Component
{
public:
   Engine(){built = false;}

   void build() {Engine::count++; built = true;}

   int get_speed() const;
   int get_max_speed() const;
   int get_max_backspeed() const;

   void get_speed(int sp);
   void get_max_speed(int sp);
   void get_max_backspeed(int sp);

   static int count;

   bool is_built(){
      return built;
   }

private:
   bool built;
   int speed_;
   int max_speed_;
   int max_backspeed_;
};

```

Листинг 15: Вспомогательный класс Weapon.

```
class Weapon: public Component
 {
 public:
    Weapon(){built = false;}
    void build() {Weapon::count++;built = true;}

    bool is_built(){return built;}

    void fire();

    static int count;

 private:
    bool built;
    int damage;
    int recharge;
    int amount_bullets;
 };

```

Листинг 16: Дополнительные классы и определение статических переменных.

```

using Ms = std::chrono::milliseconds;

 class Unit{};

 class Component{};

 int Body::count = 1;
 int Tank::count = 1;
 int Weapon::count = 1;
 int Engine::count = 1;
 int TankQueue::count = 1;

```

4 Экспериментальная часть

В данном разделе будет сравнительный анализ реализаций конвейера при разной нагруженности компонентов конвейера.

4.1 Сравнительный анализ

В эксперименте были установлены следующие условия:

1) Количество элементов 10;

2) конвейер состоит из трех этапов;

3) один этап выполняется только одним потоком;

4) идеальное время построения танка(без задержек) равно 3 секунды. Имеется в виду, что всех экспериментах сумма времени постройки всех компонетов даст 3 секунды. Например постройка корпуса, двигателя и танка может занимать 0.5, 1, 1.5 секунд или 2, 0.5, 0.5 или 2.8, 0.1, 0.1. То есть в сумме они всегда дают 3 секунды;

Первые четыре опыта проводились, чтобы показать зависимость скорости выполнения от разницы скорости работы различных составляющих.

1. В первом опыте время первой ступени варьировалась от 900 до 100 с шагом 200, время второй ступени от 1100 до 1900 с тем же шагом. Время третьей ступени константно равна 1000.

2. Во втором опыте время первой ступени варьировалась от 1100 до 1900 с шагом 200, время третьей ступени от 900 до 100 с тем же шагом. Время второй ступени константно равна 1000.

3. В третьем опыте время второй ступени варьировалась от 900 до 100 с шагом 200, время третьей ступени от 1100 до 1900 с тем же шагом. Время первой ступени константно равна 1000.

Четвертый опыт показывает наилучший и наихудший случаи: с наименьшей и наибольшей разницей

Время каждой ступени менялось так чтобы соблюдался баланс, объявленный в пункте 4. Время ожидания на мьютексе и время ожидания поступления новых танков понижены до 10, поэтому на эксперименте эти величины сказаться не должны. Поскольку все четыре опыта показывают одну и ту же зависимость, они все будут включены в одну таблицу 1. Она состоит из трёх таблиц: время работы первого/второго/третьего этапа для одного элемента и суммарное время работы первого/второго/третьего этапа. Последний пункт отражает суммарное время работы поскольку, третий этап конвейера запускается в то же время, что и самый первый, а заканчивается самым последним. Для наглядности на рис. **??** продемонстрирована зависимость разницы между минимальным и максимальным временем работы и суммарной скоростью работы конвейера.

Таблица 1: Зависимость времени работы конвейера от разницы времени работы отдельных его составляющих

Время одного выполнения первого этапа (в мс)	Время одного выполнения второго этапа (в мс)	Время одного выполнения третьего этапа (в мс)	Суммарное время работы работы этапа (в мс)	Суммарное время работы работы этапа (в мс)	Суммарное время работы работы этапа (в мс)
900	1100	1000	5005	6906	7916
700	1300	1000	4005	7711	8717
500	1500	1000	3006	8509	9510
300	1700	1000	2004	9311	10313
100	1900	1000	1008	10108	11111
1100	1000	900	6006	7013	7920
1300	1000	700	7005	8011	8716
1500	1000	500	8005	9015	9520
1700	1000	300	9004	10008	10310
1900	1000	100	10005	11014	11121
1000	900	1100	5505	6407	7910
1000	700	1300	5507	6212	8712
1000	500	1500	5505	6011	9510
1000	300	1700	5507	5813	10317
1000	100	1900	5505	5610	11118
100	100	2800	1005	1107	14709
1000	1000	1000	5505	6509	7511
2800	100	100	14505	14611	14715

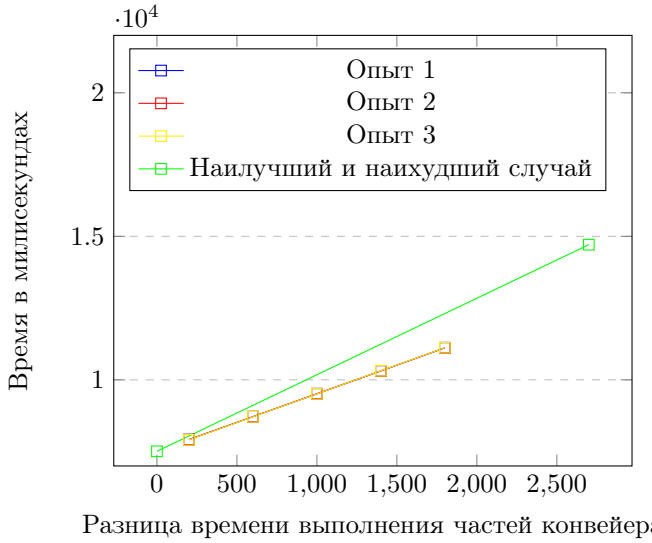

Рис. 6: График зависимости времени работы конвейера от разницы времени работы ступеней.

В следующем опыте рассматриволось зависимость скорости работы от количества потоков. Для этого время ожидания на мьютексе и ожидания новых элементов было повышено до 100мс, а количество элементов до 10. Количество потоков варьируется от 1 до 10. Количество времени на каждый этап - 1 секунда. Результаты представлены в таблице 2. Первый столбец - количество потоков. Остальные - время работы соотвествующего этапа конвейера. Результаты провизуализированы на рисунке 7.

Таблица 2: Зависимость времени работы конвейера от разницы времени работы отдельных его составляющих

Количество потоков	Суммарное время выполнения первого этапа (в мс)	Суммарное время выполнения второго этапа (в мс)	Суммарное время выполнения третьего этапа (в мс)
1	12608	13614	14716
2	6407	7408	8411
3	4304	5307	6312
4	3352	4356	5360
5	3252	4256	5259
6	2405	3405	4412
7	2307	3312	4316
8	2303	3408	4307
9	2203	3308	4306
10	2201	3409	4407

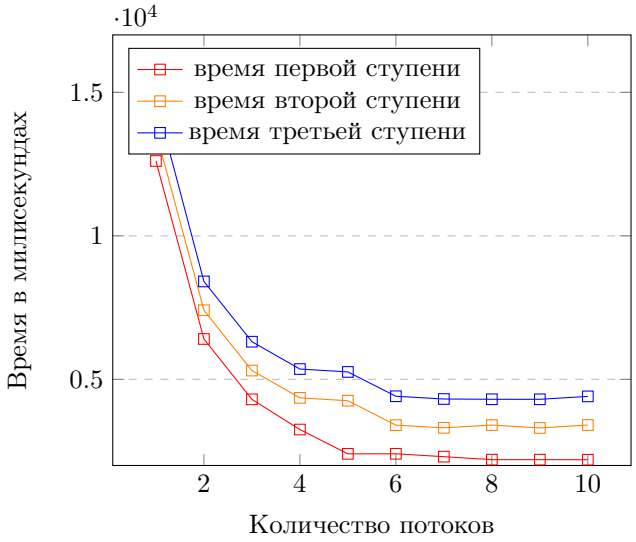

Рис. 7: График зависимости времени работы конвейера от количества потоков.

В заключительном опыте был проведен тестовый запуск конвейера с 5 танками с задержками 50мс с целью показать хронологию работы конвейера. Время работы первого этапа выставлено на 700мс, второго на 1400мс, третьего на 900мс. Полученные данные отражены на рисунке 8. Для удобства график продублирован на рис. 9

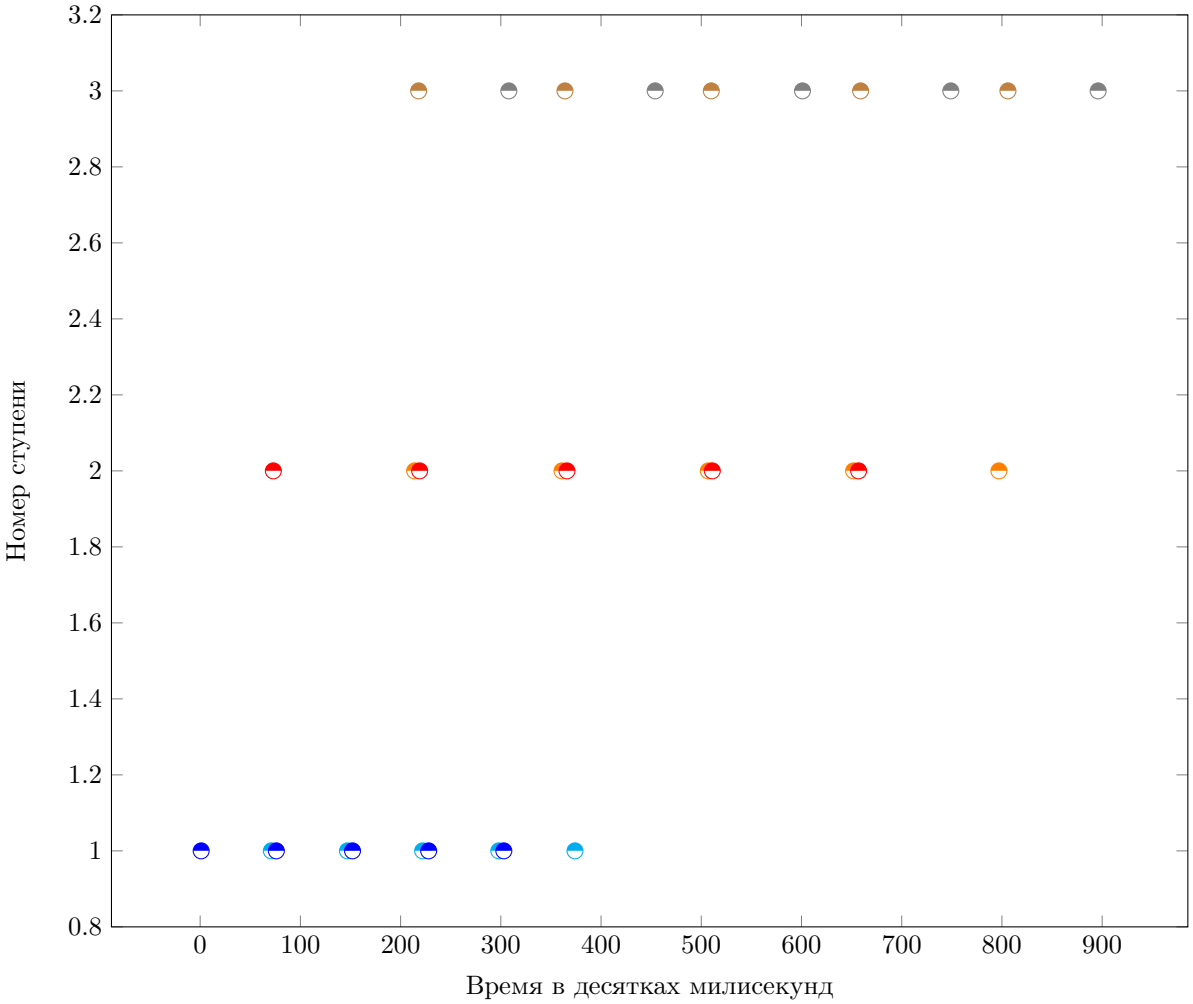

Рис. 8: Работа конвейера. Более светлым оттенком отмечено время выхода из очереди, более тёмным - время добавления в очередь

Вывод

Эксперименты показали, что конвейер работает наиболее быстро, когда время работы каждого из его составляющих примерно одинаково. Наилучшая скорость достигается, когда количество потоков равно количеству элементов, пришедших на конвейер, что ожидаемо.

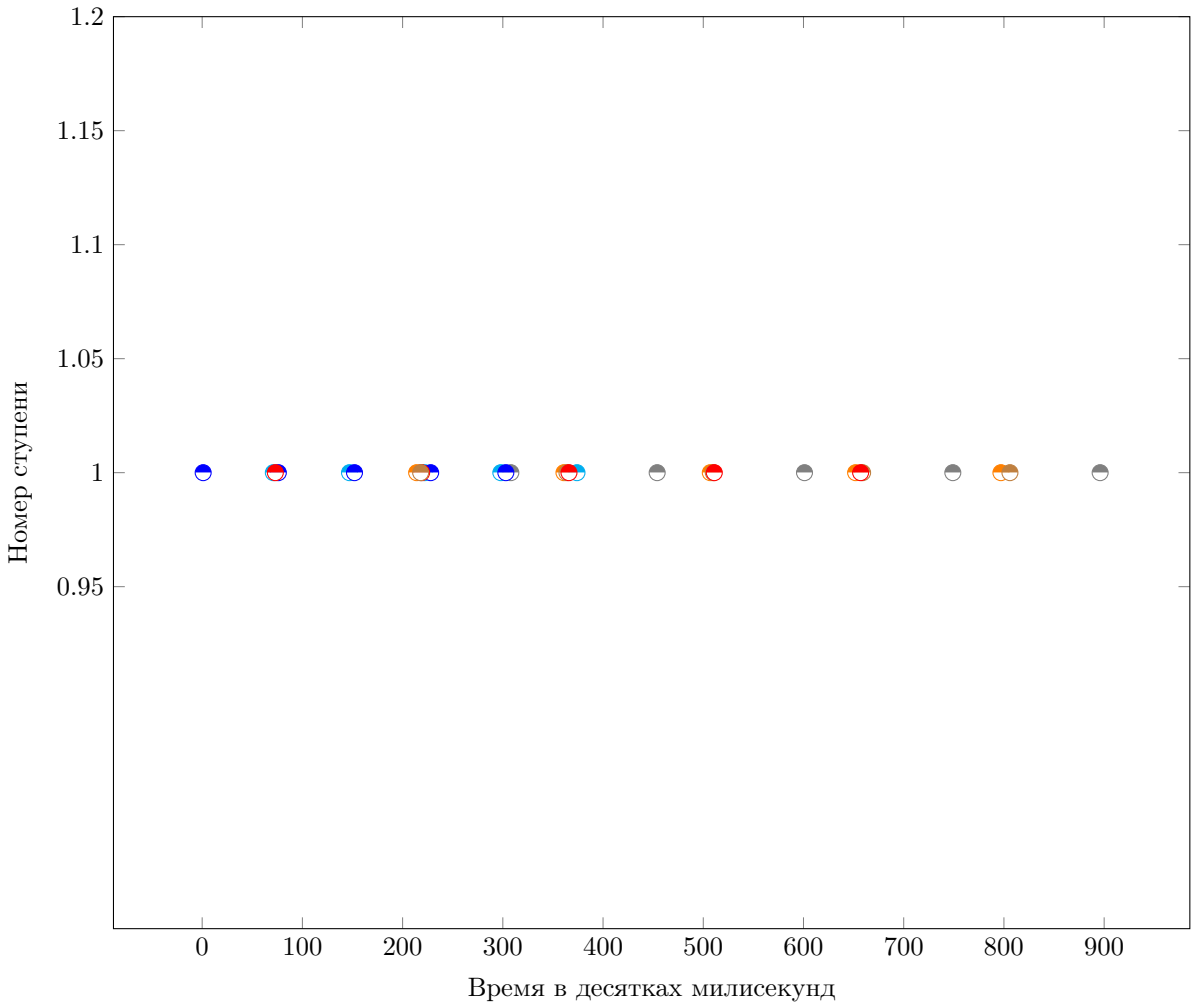

Рис. 9: Работа конвейера, где все ступени показаны на одной ветве.

Заключение

В ходе работы был изучаен и реализован вычислительный конвейер из трёх этапов. Были определены наиболее оптимальные параметры для быстрого решения: количество потоков.

Список литературы

[1] Конвейерная обработка данных https://studfiles.net/preview/1083252/page:25/.

[2] Описание RISC-процессора http://www.metods-rgrtu.ru/index.php/metods-3300-3399/309-3330.

[3] Арифметический конвейер https://studfiles.net/preview/364715/page:3/.

[4] Суперконвейер https://studopedia.su/$15_132364_superkonveyernie - protsessori.html$.

[5] Оценка производительности https://www.kazedu.kz/referat/132609/2.

[6] Формулы выбора города, обновления феромонов https://habr.com/post/105302/

[7] И.В. Ломовской. Курс лекций по языку программирования C, 2017.