МГТУ им. Баумана

Лабораторная работа №6

По курсу: "Анализ алгоритмов"

Вычислительный конвейер

Работу выполнила: Лаврова Анастасия, ИУ7-55Б

Преподаватели: Волкова Л.Л., Строганов Ю.В.

Москва, 2019

Оглавление

Введение

Целью данной лабораторной работы является применение конвейерной обработки данных. Конвейерная обработка данных может быть полезной,когда каждая операция занимает много времени. В данной работе будет рассмотрена задача "построения автомобиля а также проведения замера скорости работы работы конвейера при разной нагруженности блоков.

1 Аналитическая часть

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.
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.

1.1 Оценка производительности идеального конвейера

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

$$\tau_e = \sum_{i=1}^{n} \tau_i \tag{1.1}$$

где

n — количество последовательных этапов;

τ_i — время выполнения i-го этапа;

Быстродействие одного процессора, выполняющего только эту операцию, составит

$$S_e = \frac{1}{\tau_e} = \frac{1}{\sum_{i=1}^{n} \tau_i} \tag{1.2}$$

где

τ_e — время выполнения одной операции;

n — количество последовательных этапов;

τ_i — время выполнения i-го этапа;

Максимальное быстродействие процессора при полной загрузке конвейера составляет

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

$$\frac{n}{n/2} \leq \frac{S_{max}}{S_e} \leq n \tag{1.3}$$

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.

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

Требования к вводу:
-
Требования к программе:
Корректная работа конвейера

2.1 Разработка реализации программы

Создаются 5 очередей: 3 очереди для 3-х этапов сборки машины, 1 очередь для "пустых"и 1 очередь для собранных машин.
Запускаются потоки для каждой очереди. Функция потока выбирает метод start, который принимает на вход ссылку на следующю очередь. Также функция постоянно ожидает, когда флаг конца работы примет значение True; в противном случае функция проверяет, хранятся ли в очереди необработанные машины. Если остались необработанные машины, то mutex захватывается, достается машина из очереди, mutex освобождается. Начинается обработка машины с помощью функтора work. После выполнения "установки"компонента происходит попытка подключения к очереди следующего этапа конвейера. Ожидается следующая очередь time-for-mutex секунд, если подключиться не удается - засыпаем до тех пор, пока очередь выполнения не опустеет.
Если данный этап является "первым"(то есть первым из незавершенных), то назначем следующий этап "первым а флагу конца работы присваиваем значение True и прекращаем его работу. Если же этот этап не является "первым то процесс засыпает до тех пор, пока не поступит новая машина, после чего процесс посыпается, обрабатывает очередь, заносит машины в следующую очередь, и так до тех пор, пока этот этап не

станет "первым"и не отработает до конца, после этого к нему мы больше не будем возвращаться.

На рис. 2.1 представлена схема установки компонента в машину:

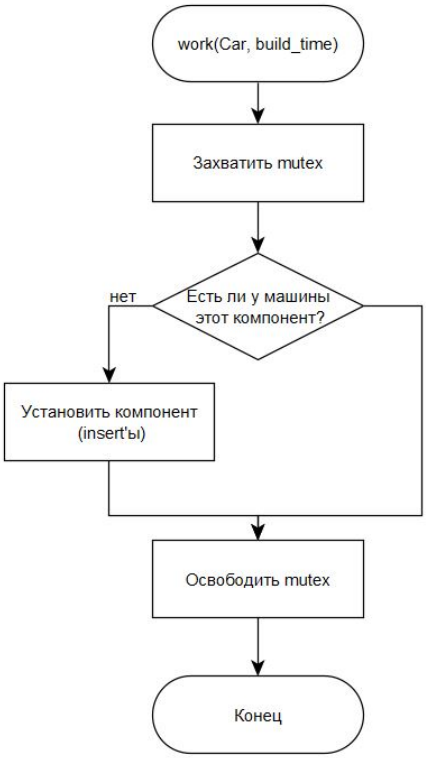


Рис. 2.1: Схема установки компонента в машину

На рис. 2.2 представлена схема работы одного этапа конвейера:

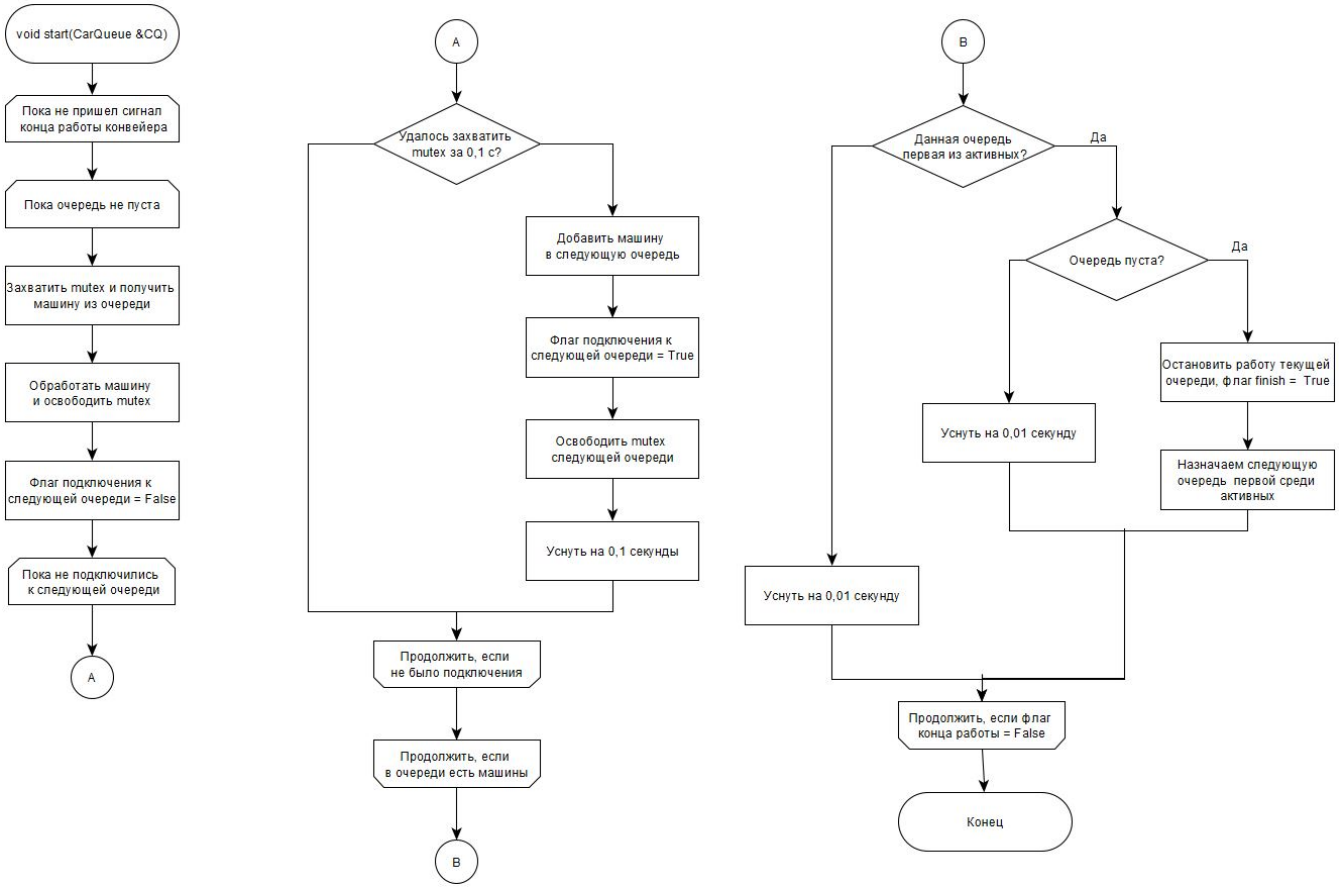


Рис. 2.2: Схема работы одного этапа конвейера

3 | Технологическая часть

3.1 Выбор ЯП

Для реализации программ я выбрала язык программирования C++, так имею большой опыт работы с ним. Среда разработки - Visual Studio.

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.

Для замера процессорного времени используется функция, возвращающая количество тиков.

"Машина"состоит из трёх компонентов: корпуса, двигателя и электроники. Один из трёх этапов сброки представлен классом CarQueue, который хранит следующие поля:

1) cars - очередь машин, 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

Листинг 3.1: Функция получения тиков

```
unsigned __int64 tick()
{
  return (__rdtsc());
}
```

3.2 Реализация вычислительного конвейера

Листинг 3.2: Функция start

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

  while (!finish) {
    finish_mutex.unlock();
    queue_mutex.lock();
    while (Cars.size() > 0) {
      times_in.push_back(clock());
      Car Car = pop();
      queue_mutex.unlock();
      Car = work(Car, build_time);
      bool connect = 0;
      while (!connect) {
        if (TQ.queue_mutex.try_lock_for(
          Ms(time_for_mutex))) {
          TQ.push(Car);
          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();
```

```
27      if (main && Cars.size() == 0) {
28        main_mutex.unlock();
29
30        finish_mutex.lock();
31        finish = 1;
32        finish_mutex.unlock();
33
34        TQ.main_mutex.lock();
35        TQ.set_main();
36        TQ.main_mutex.unlock();
37
38
39        end = clock();
40      }
41      else {
42        main_mutex.unlock();
43        std::this_thread::sleep_for(Ms(downtime));
44      }
45      finish_mutex.lock();
46    }
47    finish_mutex.unlock();
48  }
```

Листинг 3.3: Функция add-engine - встроить двигатель

```
1  Car add_engine(Car &Car, DWORD time) {
2    Car.Car_mutex.lock();
3
4    if (!Car.has_engine()) {
5      Engine engine;
6      Car.insert_engine(engine, time);
7    }
8
9    Car.Car_mutex.unlock();
10
11   return Car;
12 }
```

Листинг 3.4: Функция add-electronics - встроить электронику

```
1  Car add_Electronics(Car &Car, DWORD time) {
```

```
2   Car.Car_mutex.lock();
3
4   if (!Car.has_Electronics()) {
5     Electronics Electronics;
6     Car.insert_Electronics(Electronics, time);
7   }
8
9   Car.Car_mutex.unlock();
10
11  return Car;
12 }
```

Листинг 3.5: Функция add-body - встроить корпус

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

Листинг 3.6: Функция main

```
1 int main()
2 {
3   Car::count = 1;
4   CarQueue::count = 1;
5
6   size_t amount = 5;
7   CarQueue Cars(amount);
8
9   CarQueue queue_engine(add_engine, 90);
10  CarQueue queue_body(add_body, 30);
11  CarQueue queue_weapon(add_Electronics, 200);
12
13  CarQueue ready_Cars;
```

```
14
15    std::thread give_to_body([&Cars, &queue_body]() { Cars.
          start(queue_body); });
16    std::thread give_to_engine([&queue_body,
17      &queue_engine]() {
18      queue_body.start(queue_engine);
19    });
20
21    std::thread give_to_weapon([&queue_engine,
22      &queue_weapon]() {
23      queue_engine.start(queue_weapon);
24    });
25
26    std::thread give_to_ready([&queue_weapon,
27      &ready_Cars]() {
28      queue_weapon.start(ready_Cars);
29    });
30
31    give_to_body.join();
32    give_to_engine.join();
33    give_to_weapon.join();
34    give_to_ready.join();
35
36    std::cout << "\n\ni finished! " << ready_Cars.size() << "
          " << queue_weapon.size() << queue_engine.size() <<
         queue_body.size();
37
38    return 0;
39  }
```

Листинг 3.7: Функция CarQueue - конструктор класса для пустой очереди

```
1  CarQueue::CarQueue(size_t amount) {
2    end = begin = clock();
3    id = CarQueue::count++;
4    for (size_t i = 0; i < amount; i++) {
5      Car Car;
6      Cars.push(Car);
7    }
8    main = 1;
```

```
    finish = 0;
    work = noWork;
    time_for_mutex = 100;
    downtime = 100;
}
```

Листинг 3.8: Конструктор класса для очереди построения компонента

```
CarQueue::CarQueue(std::function<Car(Car&, DWORD)> f, DWORD
    t) {
    end = begin = clock();
    id = CarQueue::count++;
    main = 0;
    finish = 0;
    work = f;
    build_time = t;
    time_for_mutex = 100;
    downtime = 100;
}
```

Листинг 3.9: Конструктор копирования

```
CarQueue::CarQueue(const CarQueue & other) {
    id = other.id;
    Cars = other.Cars;
    work = other.work;
    finish = other.finish;
    main = other.main;
    time_for_mutex = other.time_for_mutex;
    downtime = other.downtime;
}
```

Листинг 3.10: Конструктор перемещения

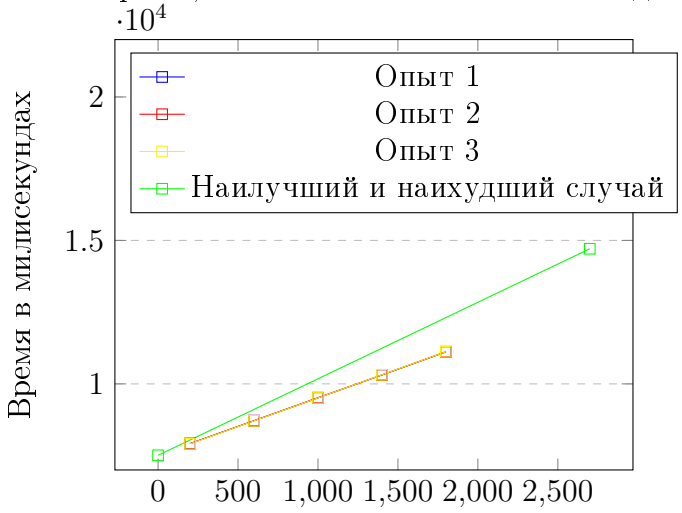
```
CarQueue::CarQueue(const CarQueue && other) {
    id = other.id;
    Cars = other.Cars;
    work = other.work;
    finish = other.finish;
    main = other.main;
    time_for_mutex = other.time_for_mutex;
```

```
8     downtime = other.downtime;
9 }
```

4 | Исследовательская часть

4.1 Постановка эксперимента

Был проведен сравнительный анализ реализаций конвейера при разной нагруженности этапов конвейера. Замеры времени проводились для 10 элементов. Идеальное время сборки машины - 3 секунды.
Проведены 3 опыта, в каждом из которых время одного этапа оставалось неизменным (равное 1000 мс), а время двух других этапов варьировалось.
Последний - четвертый, - опыт показывает пример работы конвейера с наибольшей и наименьшей разницей между этапами.
В результате получается, что последний столбец отражает суммарное время работы, так как третий этап конвейера запускается в то же время, что и самый первый, а заканчивается самым последним.



Разница времени выполнения частей конвейера

Таблица 4.1: 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 1 этапа (в мс)	Суммарное время работы работы 2 этапа (в мс)	Суммарное время работы работы 3 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

4.2 Вывод

Эксперименты показали, что конвейер работает наиболее быстро, когда время работы каждого из его составляющих примерно одинаково.

Заключение

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