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 дисциплине "Анализ алгоритмов"

Тема Конвейер

Студент Байрамгалин Я.Р.

Группа ИУ7-53Б

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

Москва — 2022 г.

Оглавление

Введение

Ранее на всех предметах в МГТУ им. Баумана рассматривались лишь алгоритмы, которые выполняются последовательно. В данной лабораторной работе предстоит погрузится в безграничный мир параллельных вычислений.

Многопоточность в современных информационных системах используется повсеместно, так как во многих случаях позволяет обрабатывать большие массивы данных за то же процессорное время.

Целью данной лабораторной работы являтеся создание серверного приложения, выполняющего запросы к базе данных, способного эффективно обрабатывать множество запросов параллельно, нахождение оптимального количества потоков для получения наилучшего результата.

Для достижения цели ставятся следующие задачи:

- разработать простое серверное приложение;
- распараллелить обработку поступающих запросов;
- сравнить время обработки запросов и максимальное количество запросов в секунду, которое способно обработать серверное приложение;
- оценить оптимальное количество обслуживающих потоков;
- провести сравнительный анализ на основе полученных экспериментально данных.

1 Аналитическая часть

Запросом будем называть сообщение (под сообщением подразумевается конечный набор байтов) в формате http запроса. Формат http выбран как самый популярный и простой формат клиент-серверного взаимодействия.

Ответом будет называть сообщение в формате http ответа.

В рамках данной лабораторной работы поставлена задача разработки серверного приложения. Серверным приложением будем называть endpoint, совершающий обработку запроса и возвращающий некоторый ответ.

Обработчик получает в теле запроса целое число - количество записей, которое необходимо вернуть. Возвращается фиксированное число государственных номеров автомобилей из базы данных.

1.1 Протокол HTTP

HTTP — протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.

Формат http запроса состоит из:

- Версии протокола.
- HTTP-метода, обычно глагола подобно GET, POST или существительного, как OPTIONS или HEAD, определяющее операцию, которую клиент хочет выполнить. Обычно, клиент хочет получить ресурс (используя GET) или передать значения HTML-формы (используя POST), хотя другие операция могут быть необходимы в других случаях.
- Пути к ресурсу.
- Заголовков с метаинформацией (опционально).
- Тела запроса, предворяемого пустой строкой (опционально).

Пример http запроса представлен в листинге 1.1.

Формат http ответа состоит из:

- Версии протокола.
- Код состояния. Наиболее популярными являются: 200 (запрос выполнен успешно), 4xx (плохой запрос), 5xx (внутренняя ошибка сервера).
- Сообщение состояния, например OK, Bad request, Internal Server Error.
- Заголовки с метаинформацией (опционально).
- Тела ответа, предворяемого пустой строкой (опционально).

Пример http ответа представлен в листинге 1.2.

```
1  POST /echo/post/json HTTP/1.1
2  Host: reqbin.com
3  Authorization: Bearer mt0dgHmLJMVQhvjpNXDyA83vA_PxH23Y
4  Content-Type: application/json
5  Content-Length: 80
6
7  {
8    "Id": 12345,
9    "Customer": "John Smith",
10   "Quantity": 1,
11   "Price": 10.00
12 }
```

Листинг 1.1: Пример http запроса

```
HTTP/1.1 201 Created
Location: http://localhost/objectserver/restapi/alerts/status/kf/12481%3
    ANCOMS
Cache-Control: no-cache
Server: libnhttpd
Date: Wed Jul 4 15:31:53 2012
Connection: Keep-Alive
Content-Type: application/json;charset=UTF-8
Content-Length: 304

{
  "entry":  {
    "affectedRows": 1,
    "keyField": "12481%3ANCOMS",
    "uri": "http://localhost/objectserver/restapi/alerts/status/kf/12481%3
    ANCOMS"
  }
}
```

Листинг 1.2: Пример http ответа

1.2 Серверное приложение

Socket — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой только сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Для взаимодействия между машинами используются адреса и порты. Адрес представляет собой 32-битную структуру для протокола IPv4, 128-битную для IPv6. Номер порта — целое число в диапазоне от 0 до 65535. Эта пара и определяет сокет («гнездо», соответствующее адресу и порту).

Для выполнения задачи построения серверного приложения необходимо читать запросы, поступающие на определенный сокет, посылать их в обработчик и затем записывать их по другому сокету, созданному для принятия ответа.

1.3 Рапараллеливание обработки запросов

Для решения поставленной задачи обработки запросов существует несколько способов организации параллельности вычислений:

- Выполнять все в одном потоке. Такой подход является неэффективным, так как пока не будет выполнен первый запрос, поступивший на обработчик следующий запрос не будет взят в работу.

- Обработка каждого запроса в отдельном потоке. Такой подход также не является эффективным, так как при высокой загруженности такого серверного приложения количество потоков в системе может стать слишком большим и значительная часть процессорного времени будет тратиться на организацию работы потоков, а не на обработку самих запросов.

- Создание системы с фиксированным числом потоков, где один поток занимается лишь чтением запросов, поступающих на сокет, а остальные обрабатывают запросы и посылают ответы. Данный подход кажется оптимальным в данной ситуации, однако трубует дополнительного анализа для поиска оптимального числа потоков.

Последний пункт может быть формализован как реализация модели пула потоков. Данная модель параллельных вычислений состоит в организации работы следующих сущностей:

- производитель, роль которого заключаются в отправке задач в общую очередь задач;

- очередь задач, роль которой заключется в накоплении задач производителя и из передаче потребителю;

- потребители, роли которых заключаются в забирании задач из очереди и их обработке.

Схему работы пула потоков более наглядно можно проследить на рисунке 1.1.

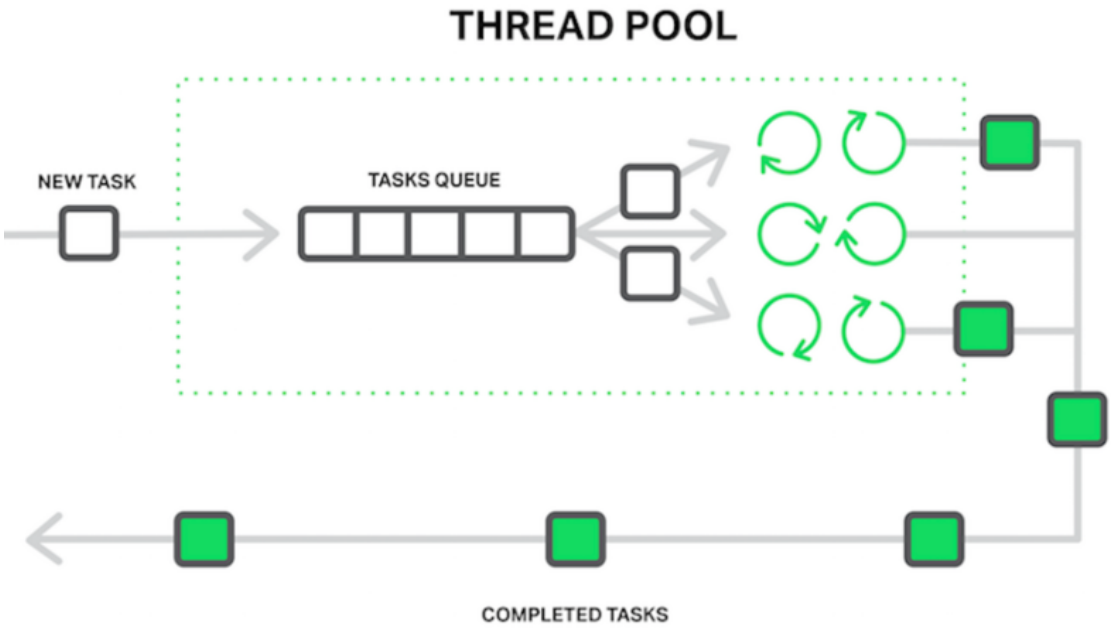

Рис. 1.1: Схема работы пула потоков

Развивая данную идею получено, что каждый потребитель может передавать ставить задачи слеующему потребителю, тем самым выступая проихводителем. Такая организация вычислений называется конвеер.

Вывод

Рассмотрены теоретические построения серверного приложения. Получены знания, достаточные для перехода к программной реализации задачи.

2 Конструкторская часть

2.1 Серверное приложение

Алгоритм работы серверного приложения можно разделить на 2 части: инициация сервера и обработка запросов.

Для инициация сервера необходимо сделать следующие:

- создать endpoint для комуникации системным вызовом socket;
- установить настройки сокета;
- соединить созданный дескриптор сокера с ассоциированным портом.

Для обработки запросов после инициации сервера необходимо бесконечно (до получения сигнала о завршении) выполнять следующие действия:

- слушать сокет системным вызовом listen;
- при получении запроса создать новое соединение системным вызовом accept;
- отправить полученный запрос и дескриптор нового соединения в очередь задач.

2.2 Очередь задач

Очередь задач — программная сущность, поддерживающая следующие операции: добавить задачу в очередь, получить задачу из очереди.

2.3 Потребители

Потребители — фиксированное число потоков, которые:

- забирают задачу из очереди;

- обрабатывают задачу;
- ставят в следующую очередь или же отправляют ответ пользователю;

забирают задачи из очереди задач

2.4 Парсинг http запросов

Задача парсинга http запросов не относится к сути выполненной лабораторной работы и поэтому подробное описание алгоритма парсинга http запросов не приводится.

2.5 Модель оценки эффективности серверного приложения

Для оценки эффективности серверного приложения, разрабатываемого в рамках данной лабораторной работы, продложены следующие метрики: время обработки запросов в персентилях 50, 66, 75, 80, 90, 95, 98, 99, 100; количество запросов в секунду, при котором сервер может стабильно работать (rps).

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, используемые технологии и реализации алгоритмов.

3.1 Требования к программному обеспечению

К программе предъявляется ряд требований:

- запрос поступает по заданному адресу сетевого соединения на порт 5002;
- запрос поступает в формате http запроса, версия протокола 1.1;
- в запросе HTTP метод — POST;
- в запросе путь к ресурсу — /api/multiply_matrix;
- в запросе указан заголовок Content-Type, его значение — text;
- в запросе указан заголовок Content-Length, его значение — длина тела запроса;
- в теле запроса располагается единственное положительное целое число - количество записей, которые требуется вернуть;
- ответ поступает в формате http ответа.
- ответ содержит зоголовок Content-Type, его значение — text;
- ответ содержит заголовок Content-Length, его значение — длина тела запроса;
- при успешной обработке запроса в ответе содержится код выполнения 200, статус OK, в теле ответа массив автомобильных номеров из базы данных;
- при ошибке соответствия формата запроса протоколу http запроса в ответе содержится код выполнения 400, статус Bad Request;

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования C++[1]. Язык позволяет управлять всеми ресурсами компьютера и позволяет делать необхоимые системные вызовы.

Тестирование корректности работы серверного приложения производилось с помощью утилиты curl[2].

Метрики, изложенные в подразделе 2.6 замерены утилитой ab - Apache HTTP server benchmarking tool[3]

3.3 Реализация алгоритмов

Для параллельной обработки запросов были разработаны два класса: BlockingQueue, TreadPool. Их реализация представлена в лимтингах 3.1, 3.2 соответственно.

Непосредственно реализация серверного приложения представлена в классе Server в листинге 3.3.

```
1  #pragma once
2
3  #include <deque>
4  #include <mutex>
5  #include <thread>
6  #include <chrono>
7  #include <condition_variable>
8
9  template<typename Task>
10 class BlockingQueue {
11   public:
12   void Add(Task task) {
13     std::lock_guard guard(mutex_);
14     queue_.push_back(std::move(task));
15     queue_not_empty_.notify_one();
16   }
17
18   Task Take() {
19     std::unique_lock lock(mutex_);
20     while (true) {
21       if (!queue_.empty()) {
```

```
22          return TakeLocked();
23        } else {
24          queue_not_empty_.wait(lock);
25        }
26      }
27    }
28
29    private:
30    Task TakeLocked() {
31      Task oldest = std::move(queue_.front());
32      queue_.pop_front();
33      return oldest;
34    }
35
36    std::deque<Task> queue_; // Guarded by mutex_
37    std::mutex mutex_;
38    std::condition_variable queue_not_empty_;
39  };
```

Листинг 3.1: Реализация блокирующей очереди

```
1   #pragma once
2
3   #include <thread>
4   #include <functional>
5   #include <deque>
6   #include <vector>
7   #include <stdexcept>
8
9
10  #include "pool/blocking_queue.hpp"
11
12  template<typename Task>
13  class StaticThreadPool {
14    public:
15    explicit StaticThreadPool(int thread_count)
16    : thread_count_(thread_count) {
17      StartWorkers();
18    }
19
20    void Submit(Task task) {
21      blocking_queue_.Add(std::move(task));
22    }
23
24    void Join() {
25      for (size_t i = 0; i < thread_count_; ++i) {
26        blocking_queue_.Add({});
27      }
28    }
```

```
29
30    private:
31    void StartWorkers() {
32      if (!pool_.empty()) {
33        throw std::runtime_error("Bad start workers call");
34      }
35
36      for (int i = 0; i < thread_count_; ++i) {
37        std::cout << "Starting worker " << i  << "\n";
38        pool_.emplace_back([this]() {
39          WorkerRoutine();
40        });
41      }
42    }
43
44    void WorkerRoutine() {
45      while(true) {
46        auto task = blocking_queue_.Take();
47        if (!task) {
48          break;
49        }
50        task();
51      }
52    }
53
54    const int thread_count_;
55    BlockingQueue<Task> blocking_queue_;
56    std::vector<std::thread> pool_;
57  };
```

Листинг 3.2: Реализация пула потоков

```
1  #pragma once
2
3  #include <iostream>
4  #include <string>
5  #include <functional>
6
7  #include "pool/thread_pool.hpp"
8  #include "components.hpp"
9
10 class Server {
11   using Task = std::function<void()>;
12   public:
13   Server();
14
15   void StartListening();
16
17   private:
```

```
18    int socket_descriptor_;
19  };
```

Листинг 3.3: Реализация сервара из файла server/server.hpp

```
1   #include "server.hpp"
2
3   #include <iostream>
4   #include <stdexcept>
5   #include <string>
6   #include <functional>
7   #include <memory>
8
9   #include <unistd.h>
10  #include <sys/socket.h>
11  #include <netinet/in.h>
12
13  #include <pqxx/pqxx>
14
15  #include "models/request.hpp"
16  #include "models/response.hpp"
17  #include "utils/parse.hpp"
18  #include "views/handler.hpp"
19  #include "pool/thread_pool.hpp"
20  #include "components.hpp"
21
22  namespace {
23
24    constexpr const int kServerPortNumber = 5002;
25
26    constexpr const int kParseThreadCount = 2;
27    constexpr const int kDbThreadCount = 8;
28    constexpr const int kResultThreadCount = 2;
29
30    inline constexpr const char kConnectionString[] =
31    "host=localhost dbname=lab_01";
32
33    using Task = std::function<void()>;
34
35    StaticThreadPool<Task> thread_pool_parse{kParseThreadCount};
36    StaticThreadPool<Task> thread_pool_db_connect{kDbThreadCount};
37    StaticThreadPool<Task> thread_pool_response{kResultThreadCount};
38
39    void ProcessResult(
40    int socket_descriptor,
41    const std::vector<std::string>& car_numbers) {
42      Response response = [&](){
43        if (car_numbers.empty()) {
44          return Response{400, "Bas request"};
```

```
45        }
46        std::string response_string = "{\n";
47          for (const auto& number : car_numbers) {
48            response_string += fmt::format("\t'{}',\n", number);
49          }
50          response_string += "}\n";
51
52        return Response{200, response_string};
53      }();
54
55      const auto raw_response = BuildResponseString(response);
56      write(socket_descriptor, raw_response.c_str(), raw_response.size());
57    }
58
59    void ProcessDb(
60    int socker_descriptor,
61    Request request) {
62      if (!request.body.has_value()) {
63        thread_pool_response.Submit([=](){
64          ProcessResult(socker_descriptor, {});
65        });
66      }
67
68      int count_from_requests;
69      try {
70        count_from_requests = std::stoi(request.body.value());
71      } catch (const std::exception& e) {
72        thread_pool_response.Submit([=](){
73          ProcessResult(socker_descriptor, {});
74        });
75        return;
76      }
77
78      std::vector<std::string> result;
79      const auto query = fmt::format(
80      "select state_number "
81      "from public.cars "
82      "order by state_number "
83      "limit {}", count_from_requests);
84      try {
85        pqxx::connection db_connection(kConnectionString);
86        pqxx::work transaction{db_connection};
87        for (auto [number] : transaction.stream<std::string>(query)) {
88          result.push_back(number);
89        }
90      } catch (const std::exception& e) {
91        std::cout << e.what();
92        return;
```

```
93      }
94
95      thread_pool_response.Submit([=](){
96        ProcessResult(socker_descriptor, result);
97      });
98    }
99
100   void ProcessParse(int socket_descriptor) {
101     constexpr const size_t buffer_size = 524288;
102     std::unique_ptr<char[]> buffer = std::make_unique<char[]>(buffer_size)
      ;
103     bzero(buffer.get(), buffer_size);
104
105     if (recv(socket_descriptor, buffer.get(),
106     buffer_size, 0) == -1) {
107       throw std::runtime_error("Read error");
108     }
109
110     Request request{};
111     try {
112       std::cout << buffer.get();
113       request = utils::Parse(buffer.get(), utils::To<Request>());
114     } catch (std::exception& error) {
115       std::cerr << "Error occurred while handling request : " << error.
      what();
116     }
117
118     thread_pool_db_connect.Submit([=](){
119       ProcessDb(socket_descriptor, request);
120     });
121   }
122
123 }
124
125 Server::Server()
126 : socket_descriptor_(socket(AF_INET, SOCK_STREAM, 0)) {
127   if (socket_descriptor_ == -1) {
128     throw std::runtime_error("Socker error");
129   }
130
131   int enable = 1;
132   const auto opt_rc = setsockopt(socket_descriptor_, SOL_SOCKET,
      SO_REUSEADDR,
133   &enable, sizeof(int));
134   if (opt_rc == -1) {
135     throw std::runtime_error("Setopt error");
136   }
137
```

```
138   sockaddr_in address{};
139   address.sin_family = AF_INET;
140   address.sin_addr.s_addr = INADDR_ANY;
141   address.sin_port = htons(kServerPortNumber);
142
143   auto* converted_address = (sockaddr*)&address;
144   if (bind(socket_descriptor_, converted_address, sizeof(address)) < 0) {
145     throw std::runtime_error("Bind error");
146   }
147
148   std::cout << "Server is ready to accept connections\n";
149 }
150
151 void Server::StartListening() {
152   listen(socket_descriptor_, 100);
153
154   sockaddr_in client_address{};
155   socklen_t client_length = sizeof(client_address);
156   std::cout << "Listening on port " << kServerPortNumber << "\n";
157
158   while (true) {
159     const auto new_sfd = accept(
160     socket_descriptor_, (sockaddr*)&client_address, &client_length);
161     if (new_sfd < 0) {
162       throw std::runtime_error("Accept error");
163     }
164     std::this_thread::sleep_for(std::chrono::milliseconds(1));
165     thread_pool_parse.Submit([new_sfd]() {
166       ::ProcessParse(new_sfd);
167     });
168   }
169 }
```

Листинг 3.4: Реализация сервара и конвеера из файла server/server.cpp

3.4 Тестрование

Рассмотрены несколько тестовых случае, указанных ниже.

- запрос из листинга 3.5 - ответ из листинга 3.6

```
1   post /api/multiply_matrix HTTP/1.0
2   Host: localhost:5002
3   User-Agent: curl/7.84.0
4   Accept: */*
5   Content-Type: text
```

```
6   Content-Length: 25
7
8   2 2
9   5 7
10  9 1
11  2 2
12  7 4
13  1 -5
14
```

Листинг 3.5: Запрос, проверяющий правильность ответа на корректных данных

- запрос из листинга 3.5 - ответ из листинга 3.6

```
1   HTTP/1.1 200 OK
2   Content-Length: 25
3   Content-Type: text
4
5   2 2
6   42 -15
7   64 31
8
```

Листинг 3.6: Ожидаемый ответ на запрос из листинга 3.5

Вывод

Разработано серверное приожение, удовлетворяющее изложенным требованиям

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, следующие.

- Операционная система: macOs Monterey 12.4[2].
- Память: 16 Гбайт.
- Процессор: 2,6 ГГц 6-ядерный процессор Intel Core i7[3].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Анализ метрик

В таблице 4.1 представлена зависимость времени выполнения запросов в персентелях от количества потоков в системе.

В таблице 4.2 представлено количетсов запросов в секунду, которое сервер может успешно обработать.

Таблица 4.1: Количество запросов в секунду, которое может обработать сервер

Потоков, ед.	50%, мкс	90%, мкс	95%, мкс	98%, мкс	100%, мкс
2	3956	4011	4162	4254	4287
6	904	947	964	1002	1075
12	734	792	901	932	1154
23	710	790	810	854	859
31	726	885	900	1023	1119

Таблица 4.2: Количество запросов в секунду, которое может обработать сервер

Потоков, ед.	rps
2	20
6	32
12	111
23	198
31	131

Вывод

Оптимальным является количествтво потоков, соответствующее количеству виртуальных ядер.

Заключение

В ходе выполнения лабораторной работы цель достагнута. Пешены следующие задачи:

- разработано серверное приложение;
- проанализирована зависимость времени ответа от числа потоков в системе;
- проанализирована зависимость количества обрабатываемых запросов от числа потоков в системе;
- оценено оптимальное количество обслуживающих потоков;
- проведен сравнительный анализ на основе полученных экспериментально данных.
- подготовлен отчет о лабораторной работе;

Список использованных источников

[1] C++ language [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/language (дата обращения: 04.10.2022).

[2] macos Monterey 12.03 [Электронный ресурс]. Режим доступа: https://www.apple.com/ru/macos/monterey/ (дата обращения: 04.10.2022).

[3] Процессор Intel® Core™ i5-1135G7 [Электронный ресурс]. Режим доступа: https://www.intel.ru/content/www/ru/ru/products/sku/208658/intel-core-i51135g7-processor-8m-cache-up-to-4-20-ghz/specifications.html (дата обращения: 04.10.2022).