НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра параллельных вычислительных технологий

ОТЧЕТ ПРИНЯТ

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_ / \_\_\_\_\_\_\_\_\_\_\_ /

(подпись) (расшифровка)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_ г.

**ОТЧЕТ**

по лабораторной работе № 1

Программирование независимых потоков

по дисциплине Параллельное программирование

Студенты гр. ПМИ-02

Дюков Богдан Витальевич,

Сидоров Даниил Игоревич

Новосибирск-2024

1. **Цель работы**

Познакомиться с базовыми возможностями многопоточного программирования; научиться работать с потоками, не имеющими информационных зависимостей.

1. **Использованная вычислительная система**

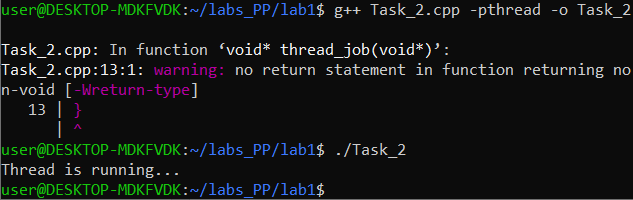
| Описание системы | |
| --- | --- |
| Аппаратная конфигурация | ЦП Intel Core i7-11700KF @ 3.60GHz (8 ядер и 16 потоков, гипертрединг включен), ОЗУ 32 ГБ |
| Программная конфигурация | ОС Ubuntu 22.04.3 LTS, компилятор g++ v. 11.4.0 |

1. **Ход работы**

1. Изучили теоретическую часть из раздела 1 настоящего пособия.

2. Записали программу из примера 1. Скомпилировали ее и проверили корректность

работы.



3. Доработали программу с целью поддержки *n* потоков.

Для поддержки *n* потоков в основном потоке создается массив поточных идентификаторов (*pthread\_t*). Размерность задается пользователем. После чего циклично вызывается функция создания присоединяемого потока (*pthread\_crea*t*e()*), благодаря чему потоки начинают исполнять поточную функцию. Создав все потоки, запускается цикл, ожидающий завершения потоков (функция *pthread\_join()*). Только после этого программа завершает свое выполнение.

5. Добавили в программу возможность запуска потоков с разными атрибутами.

Для добавления этой возможности, в *n*-поточной программе (в основном потоке) создается массив с атрибутными переменными типа *pthread\_attr\_t*. В цикле, перед созданием очередного присоединяемого потока, инициализируется соответствующая атрибутная переменная (*pthread\_attr\_ init()*). Было решено в каждом потоке изменять размерность стека. Теперь она зависит от текущего индекса в цикле (*(i+1)\*1024\*1024 байт*). Таким образом, после инициализации атрибутной переменной и задания соответствующего размер стека (*setstacksize()*), создаваемый поток получает атрибуты с обновленным размером стека.

6. Добавили в программу возможность передавать в поток сразу несколько параметров.

В функцию потока было решено передавать указатель на область, содержащую следующие параметры: индекс поточного идентификатора в массиве и атрибуты потока. Для этого создается структура, содержащая описанные параметры. В основном потоке создается массив, каждый элемент которого имеет тип созданной структуры. В цикле, когда для очередного потока уже проинициализирована атрибутная переменная и задан размер стека, происходит заполнение соответствующей структуры, после чего создаваемый поток получает возможность использовать её параметры.

7. Добавили в функцию потока возможность вывода информации о всех параметрах

потока, с которыми он был создан

После выполнения предыдущих пунктов каждая поточная функция может после простых действий вывести переданные ей параметры.

Текст программы, выполняющей пункты 3, 5, 6, 7:

#include <cstdlib>

#include <iostream>

#include <cstring>

#include <pthread.h>

#include <vector>

#include <sstream>

using namespace std;

const int NUM\_OF\_THREADS\_DEFAULT = 4;

// Структура для передачи параметров в поток

struct ThreadParams

{

int index; // Индекс потока

     pthread\_attr\_t attr; // Атрибуты потока

};

// Функция получения типа потока

int get\_detachstate(pthread\_attr\_t\* attr)

{

int detachstate;

pthread\_attr\_getdetachstate(attr, &detachstate);

return detachstate;

}

// Функция получения размера защиты

size\_t get\_guardsize(pthread\_attr\_t\* attr)

{

size\_t guardsize;

pthread\_attr\_getguardsize(attr, &guardsize);

return guardsize;

}

// Функция получения адреса и размера стека

pair<void\*, size\_t> get\_stack\_attributes(pthread\_attr\_t\* attr)

{

     void\* stackaddress;

     size\_t stacksize;

     pthread\_attr\_getstack(attr, &stackaddress, &stacksize);

     return {stackaddress, stacksize};

}

// Функция, выполняемая потоком

// Выводит информацию о потоке и его атрибутах

void \*thread\_job(void \*arg)

{

ThreadParams\* params = static\_cast<ThreadParams\*>(arg);

pair<void\*, size\_t> stack\_attr = get\_stack\_attributes(&params->attr);

ostringstream out;

out << "Thread " << pthread\_self() << " with index " << params->index << " is running...\n"

<< "Attribute values: \n"

<< "\tDetach state = " << (get\_detachstate(&params->attr) == PTHREAD\_CREATE\_DETACHED ? "Detached" : "Joinable") << "\n"

<< "\tGuard size = " << get\_guardsize(&params->attr) << " bytes \n"

<< "\tStack address = " << stack\_attr.first << "\n"

<< "\tStack size = " << stack\_attr.second << " bytes \n\n";

cout << out.str();

     return NULL;

}

int main(int argc, char \*argv[])

{

int num\_of\_threads = (argc > 1 ? stoi(argv[1]) : NUM\_OF\_THREADS\_DEFAULT);

     vector<pthread\_t> threads(num\_of\_threads);

vector<pthread\_attr\_t> thread\_attrs(num\_of\_threads);

vector<ThreadParams> thread\_params(num\_of\_threads);

     int status;

     for(int i = 0; i < num\_of\_threads; i++)

{

// Инициализируем переменную, содержащую сведения о значениях атрибутов потока

status = pthread\_attr\_init(&thread\_attrs[i]);

if(status != 0)

{

             cerr << "Cannot create thread attribute: " << strerror(status) << endl;

             exit(-1);

        }

// Устанавливаем стековый размер, зависящий от индекса

         status = pthread\_attr\_setstacksize(&thread\_attrs[i], (i+1)\*1024\*1024);

if(status != 0)

{

             cerr << "Setting stack size attribute failed: " << strerror(status) << endl;

pthread\_attr\_destroy(&thread\_attrs[i]);

             exit(-1);

         }

// Заполняем параметры потока

thread\_params[i] = {i, thread\_attrs[i]};

// Создаем поток и сразу после удаляем переменную, связанную с атрибутами

         status = pthread\_create(&threads[i], &thread\_attrs[i], thread\_job, &thread\_params[i]);

         if(status != 0)

{

             cerr << "Cannot create a thread: " << strerror(status) << endl;

             exit(-1);

         }

     }

// Ожидаем завершения всех потоков

for(int i = 0; i < num\_of\_threads; i++)

{

status = pthread\_join(threads[i], NULL);

pthread\_attr\_destroy(&thread\_attrs[i]);

if(status != 0)

{

             cerr << "Cannot join a thread: " << strerror(status) << endl;

             exit(-1);

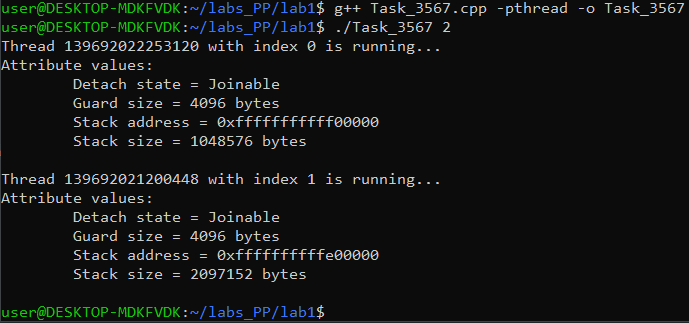
         }

     }

return 0;

}

Результаты выполнения программы:



4. Оценили стоимость запуска одного потока операционной системой.

Пусть есть функция, выполняющая суммирование чисел от 0 до заданного пользователем числа операций. Будем выполнять эту функцию двумя разными способами. Первый способ – функцию будет выполнять один раз порожденный поток и один раз основной поток. Второй способ – функцию будет выполнять два раза только основной поток. Найдем такое число операций, чтобы время выполнения первым способом было меньше, чем вторым. Тогда мы сможем сказать, что порождение потока оправданно.

Текст программы:

#include <iostream>

#include <cstring>

#include <pthread.h>

#include <vector>

#include <sstream>

#include <iomanip>

#include "../timer.h"

using namespace std;

const int NUM\_OPERATIONS\_DEFAULT = 10000;

// Функция, выполняемая потоком

// Суммирует числа от 0 до num\_operations

void \*sum\_numbers(void \*arg)

{

int num\_operations = \*((int\*) arg);

     long long sum = 0;

     for(int i = 0; i < num\_operations; i++)

{

       sum += i;

     }

     return NULL;

}

// Функция для расчета времени выполнения с дополнительным потоком

double calculate\_execution\_time\_multi\_thread(int num\_operations)

{

     pthread\_t thread;

     int status;

// Засекаем время

Timer timer;

// Выполняем суммирование в дополнительном потоке

     status = pthread\_create(&thread, NULL, sum\_numbers, (void \*) &num\_operations);

     if(status != 0)

     {

         throw runtime\_error("Cannot create a thread: " + string(strerror(status)));

     }

// Выполняем суммирование в основном потоке

     sum\_numbers((void \*) &num\_operations);

// Ожидаем завершения созданного потока

     status = pthread\_join(thread, NULL);

if(status != 0)

     {

        throw runtime\_error("Cannot join a thread: " + string(strerror(status)));

     }

// Сохраняем время выполнения

     auto time = timer.elapsed();

     return time;

}

// Функция для расчета времени выполнения в однопоточном режиме

// Засекаем время и дважды вызываем функцию суммирования основным потоком

double calculate\_execution\_time\_single\_thread(int num\_operations)

{

Timer timer;

sum\_numbers((void \*) &num\_operations);

     sum\_numbers((void \*) &num\_operations);

     auto time = timer.elapsed();

     return time;

}

// Выводим времена расчетов в многопоточном и однопоточном режимах

int main(int argc, char \*argv[])

{

int num\_operations = (argc > 1 ? std::stoi(argv[1]) : NUM\_OPERATIONS\_DEFAULT);

     try

     {

         cout << "Execution multi thread time:  " << std::fixed << std::setprecision(8) << calculate\_execution\_time\_multi\_thread(num\_operations) << endl;

cout << "Execution single thread time: " << std::fixed << std::setprecision(8) << calculate\_execution\_time\_single\_thread(num\_operations) << endl;

     }

     catch (const runtime\_error& e)

     {

         cerr << e.what() << endl;

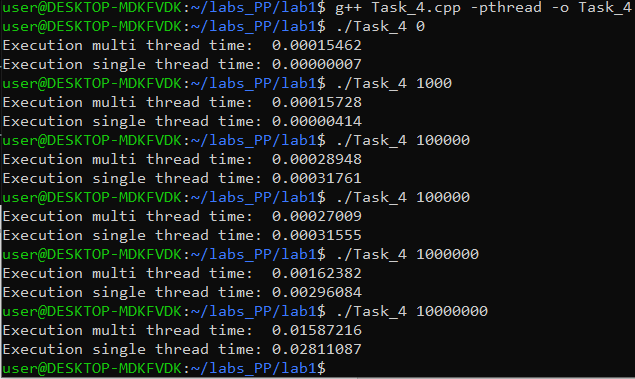
exit(-1);

     }

     return 0;

}

Результаты выполнения программы:



В нашей реализации только при количестве операций > ~100 000 использование дополнительного потока можно назвать оправданным. Начиная с этого значения, однопоточное выполнение начинает проигрывать по времени.

8. Разработали программу, которая обеспечивает параллельное применение заданной

функции к каждому элементу массива.

Имеем 3 функции, которые будут применяться к элементам массива: сложение, вычитание и умножение. Пользователь будет передавать программе следующие параметры: размерность массива, число потоков, применяемая функция. Дополнительно пользователь должен передать слагаемое/вычитаемое/множитель в зависимости от функции.

Содержимое массива до модификаций - возрастающая последовательность от 1 до заданной пользователем размерности массива. Простыми словами опишем алгоритм работы в зависимости от пользовательского ввода. Если переданная размерность массива:

* совпадает с числом потоков, то каждый из потоков выполняет функцию для соответствующего единственного элемента массива;
* меньше числа потоков, то лишние потоки “отбрасываются”, а способ обработки массива аналогичен первой ситуации;
* больше числа потоков, то надо сделать так, чтобы каждый из потоков обработал число элементов, равное целому от деления размерности массива на число потоков. Это минимум для каждого потока. Если при делении имеется остаток, то часть потоков, количество которых определяется значением остатка, получит дополнительный элемент в массиве для обработки.

Текст программы:

#include <iostream>

#include <vector>

#include <functional>

#include <map>

#include <cstdlib>

#include <ctime>

#include <pthread.h>

#include <numeric>

using namespace std;

// Значения по умолчанию

const int VECTOR\_SIZE\_DEFAULT = 10;

const int NUM\_OF\_THREADS\_DEFAULT = 4;

const int VALUE\_DEFAULT = 2;

const string OPERATION\_DEFAULT = "add";

// Определение функций для выполнения операций

void add(int& n, int value) { n += value; }

void subtract(int& n, int value) { n -= value; }

void multiply(int& n, int value) { n \*= value; }

// Сопоставление операций с их функциями

map<string, function<void(int&, int)>> operations =

{

{"add", add}, {"sub", subtract}, {"mul", multiply}

};

// Структура для передачи параметров в поток

struct ThreadParams

{

vector<int>\* numbers;  // Указатель на вектор чисел

     int begin; // Начало диапазона

     int end; // Конец диапазона

     function<void(int&, int)> func; // Функция для операции над элементами

     int value; // Значение, используемое в функции

};

// Функция, выполняемая потоком

// Применяет выбранную арифметическую операцию элементам сегмента вектора

void\* apply\_to\_segment(void\* arg)

{

     ThreadParams\* params = (ThreadParams\*)arg;

     for (int i = params->begin; i <= params->end; i++)

{

       params->func((\*params->numbers)[i], params->value);

     }

     return NULL;

}

// Функция для вывода вектора

void print\_vector(vector<int>& numbers)

{

int vector\_size = numbers.size();

for (int i = 0; i < vector\_size; i++)

{

cout << numbers[i] << ' ';

}

cout << '\n';

}

int main(int argc, char \*\*argv)

{

// Получение параметров из аргументов командной строки

     int vector\_size = (argc > 1 ? stoi(argv[1]) : VECTOR\_SIZE\_DEFAULT);

int num\_of\_threads = (argc > 2 ? stoi(argv[2]) : NUM\_OF\_THREADS\_DEFAULT);

int value = (argc > 3 ? stoi(argv[3]) : VALUE\_DEFAULT);

function<void(int&, int)> operation = (argc > 4 && operations.count(argv[4]) > 0) ? operations.at(argv[4]) : operations.at(OPERATION\_DEFAULT);

// Инициализация вектора чисел значениями от 1 до vector\_size

     vector<int> numbers(vector\_size);

iota(numbers.begin(), numbers.end(), 1);

// Печатаем содержимое вектора до операций

print\_vector(numbers);

// При превышении числа потоков над размерностью вектора отбрасываем лишние потоки

num\_of\_threads = num\_of\_threads > vector\_size ? vector\_size : num\_of\_threads;

// Вычисляем целое и остаток от деления размерности вектора на число потоков

     int quotient = vector\_size / num\_of\_threads;

int remainder = vector\_size % num\_of\_threads;

    vector<pthread\_t> threads(num\_of\_threads);

     vector<ThreadParams> threadParams(num\_of\_threads);

// Создаем потоки, предварительно вычисляя для каждого соответствующие сегменты вектора

     for (int i = 0; i < num\_of\_threads; ++i)

{

         int begin = i \* quotient + min(i, remainder);

int end = begin + quotient - (i < remainder ? 0 : 1);

         threadParams[i] = {&numbers, begin, end, operation, value};

        pthread\_create(&threads[i], NULL, apply\_to\_segment, &threadParams[i]);

     }

// Ожидаем завершения всех потоков

     for (auto& thread : threads)

{

         pthread\_join(thread, NULL);

     }

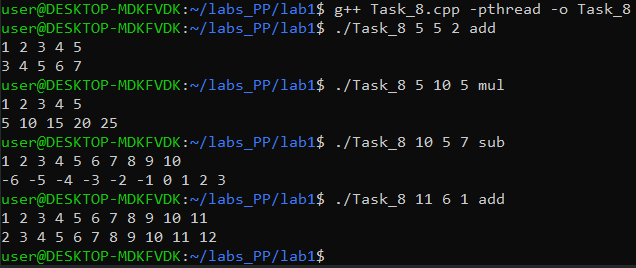
// Печатаем содержимое вектора после операций

     print\_vector(numbers);

     return 0;

}

Результаты выполнения программы (для удобства порядок передачи параметров в программу: размерность массива, число потоков, слагаемое/вычитаемое/множитель, применяемая функция):



Независимо от того, как относятся друг к другу размерность массива и число потоков, результат всегда корректен.

9. Доработали однопоточную версию сервера, доступную по адресу:

<http://rosettacode.org/wiki/Hello_world/Web_server#C>.

Обработка каждого запроса выполняется в отдельном потоке: при получении запроса создается новый поток для его обработки, после отправки результата клиенту поток завершает свою работу. Соединение с клиентом закрывается сразу после обработки запроса.

Сразу же обеспечили возможность задавать размерность стека. Если при запуске программы передается параметр 0.5, то размер стека для потоков устанавливается в 512 Кбайт, 1 – 1 Мбайт, 2 – 2 Мбайт. Также обеспечили возможность выбора: выдавать клиентам ответ вместе с версией PHP или без нее. Все эти параметры передаются в программу с помощью консольного ввода.

Текст программы:

#include <iostream>

#include <cstdlib>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <pthread.h>

#include <cstring>

#include <functional>

using namespace std;

const size\_t STACKSIZE\_DEFAULT = 2\*1024\*1024;

string response\_start = "HTTP/1.1 200 OK\r\n"

   "Content-Type: text/html; charset=UTF-8\r\n\r\n"

   "<!DOCTYPE html><html><head><title>Bye-bye baby bye-bye</title>"

   "<style>body { background-color: #111 }"

   "h1 { font-size:4cm; text-align: center; color: black;"

   " text-shadow: 0 0 2mm red}</style></head>"

   "<body><h1>Goodbye, world!";

string response\_end = "</h1></body></html>\r\n";

// Структура для передачи параметров в поток

struct ThreadParams

{

int client\_fd; // Файловый дескриптор, связанный с клиентом

     function<string()> func; // Функция, которая выдает html-код

};

// Функция для получения версии PHP

string get\_php\_version()

{

     FILE\* pipe = popen("php -r 'echo phpversion();'", "r");

     if (!pipe)

     {

         cerr << "Couldn't start PHP." << endl;

         return "";

     }

     char buffer[128];

     string phpResponse = "";

     while(!feof(pipe))

     {

         if(fgets(buffer, 128, pipe) != NULL)

{

phpResponse += buffer;

}

     }

     pclose(pipe);

     return phpResponse;

}

// Функция, возвращающая исходный html-код

string get\_normal\_response()

{

return response\_start + response\_end;

}

// Функция, возвращающая html-код с дополнительной информацией о версии php

string get\_response\_with\_php()

{

return response\_start + "<p>PHP Version: " + get\_php\_version() + "</p>" + response\_end;

}

// Функция, выполняемая потоком

// Отправляет результат клиенту и завершает свою работу

void \*handle\_client(void \*arg)

{

ThreadParams\* params = (ThreadParams\*)arg;

int client\_fd = params->client\_fd;

char buffer[1024];

int bytes\_received = recv(client\_fd, buffer, sizeof(buffer), 0);

string response = params->func();

write(client\_fd, response.c\_str(), response.size());

close(client\_fd);

delete params;

return NULL;

}

int main(int argc, char \*argv[])

{

size\_t stacksize = (argc > 1 ? static\_cast<size\_t>(stod(argv[1]) \* 1024 \* 1024) : STACKSIZE\_DEFAULT);

function<string()> func;

int func\_number;

cout << "Please choose the execution method for the program:\n"

      << "\t0 - Normal execution\n"

      << "\t1 - PHP execution\n"

      << "Enter 0 or 1: ";

cin >> func\_number;

cout << endl << "Server listening on port 8080..." << endl;

// Выбор функции в зависимости от ввода пользователя

func = (func\_number == 0) ? get\_normal\_response : get\_response\_with\_php;

     int one = 1;

     sockaddr\_in svr\_addr;

     sockaddr\_in cli\_addr;

     socklen\_t sin\_len = sizeof(cli\_addr);

     int sock = socket(AF\_INET, SOCK\_STREAM, 0);

     if (sock < 0)

     {

         throw runtime\_error("Can't open socket");

     }

     setsockopt(sock, SOL\_SOCKET, SO\_REUSEADDR, &one, sizeof(int));

     int port = 8080;

     svr\_addr.sin\_family = AF\_INET;

     svr\_addr.sin\_addr.s\_addr = INADDR\_ANY;

     svr\_addr.sin\_port = htons(port);

     if (bind(sock, (struct sockaddr \*) &svr\_addr, sizeof(svr\_addr)) == -1)

     {

         close(sock);

         throw runtime\_error("Can't bind");

     }

     listen(sock, 5);

     while (1)

     {

// Инициализируем параметры, которые будут переданы в поточную функцию

ThreadParams\* params = new ThreadParams;

        params->client\_fd = accept(sock, (struct sockaddr \*) &cli\_addr, &sin\_len);

params->func = func;

         if (params->client\_fd == -1)

         {

             cerr << "Can't accept\n";

delete params;

             continue;

         }

         pthread\_t thread;

pthread\_attr\_t attr;

// Инициализируем переменную, содержащую сведения о значениях атрибутов потока

int status = pthread\_attr\_init(&attr);

if(status != 0)

{

             cerr << "Cannot create thread attribute: " << strerror(status) << endl;

             delete params;

             continue;

         }

// Устанавливаем стековый размер, зависящий от ввода пользователя

status = pthread\_attr\_setstacksize(&attr, stacksize);

if(status != 0)

{

             cerr << "Setting stack size attribute failed: " << strerror(status) << endl;

pthread\_attr\_destroy(&attr);

             delete params;

             continue;

         }

// Создаем поток и сразу после удаляем переменную, связанную с атрибутами

         status = pthread\_create(&thread, &attr, handle\_client, params);

pthread\_attr\_destroy(&attr);

         if(status != 0)

         {

             cerr << "Cannot create a thread: " << strerror(status) << endl;

delete params;

             continue;

         }

// Изменяем тип потока на отсоединенный

status = pthread\_detach(thread);

if(status != 0)

         {

             cerr << "Cannot detach a thread: " << strerror(status) << endl;

             continue;

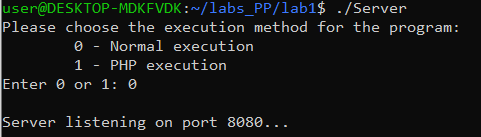
         }

     }

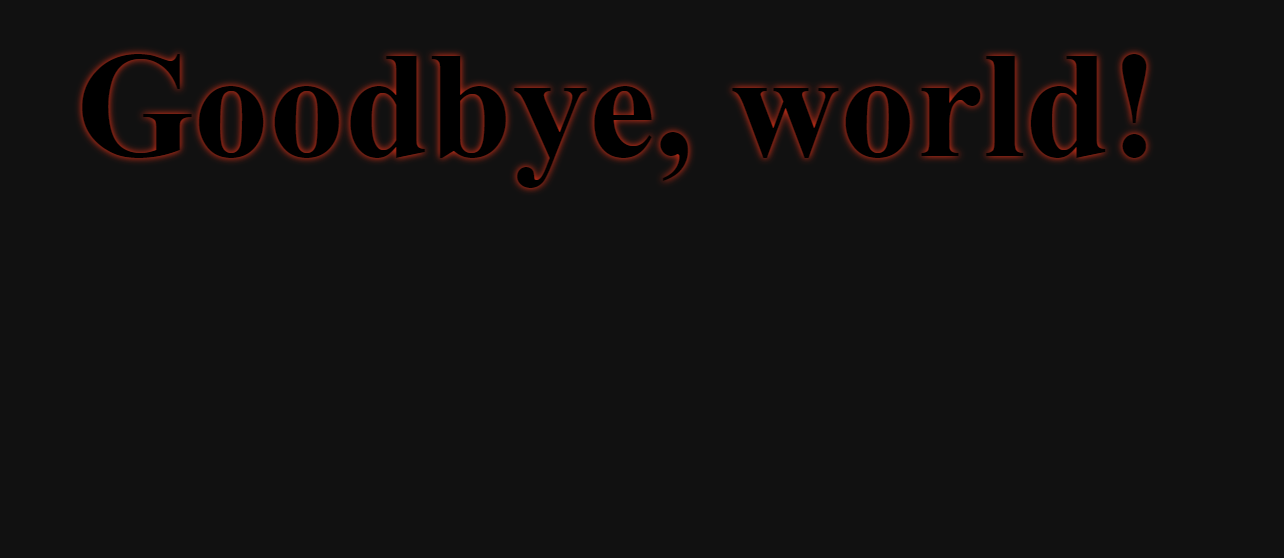
     pthread\_exit(NULL);

}

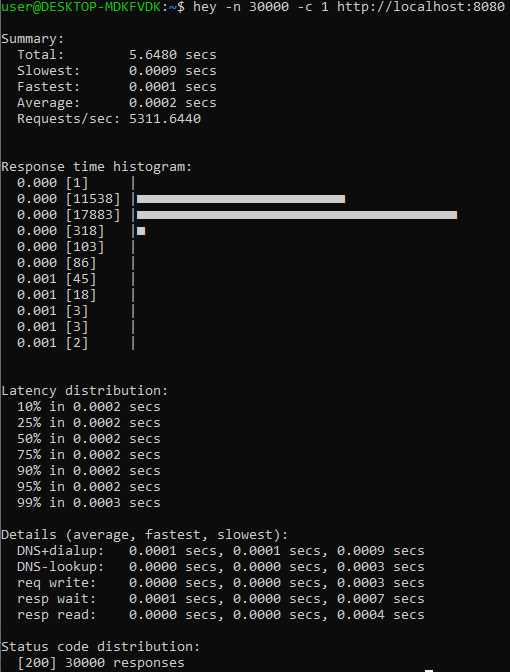
Результаты выполнения:



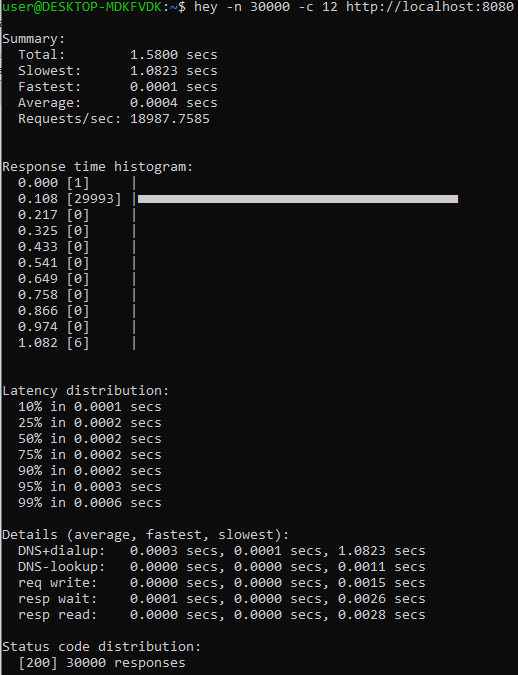
Ответ сервера при выборе ответа без PHP версии:



Оценим производительность веб-сервера (без отправки PHP-версии) с помощью утилиты hey. Размер стека оставим по умолчанию – 2 МБ. Утилита отправит 30 000 запросов без одновременных соединений:

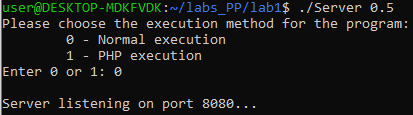


Сервер справился с этими запросами за ~5.6 секунд. Увеличивая количество одновременных запросов до определенного момента (около 12), можно значительно ускорить время выполнения:

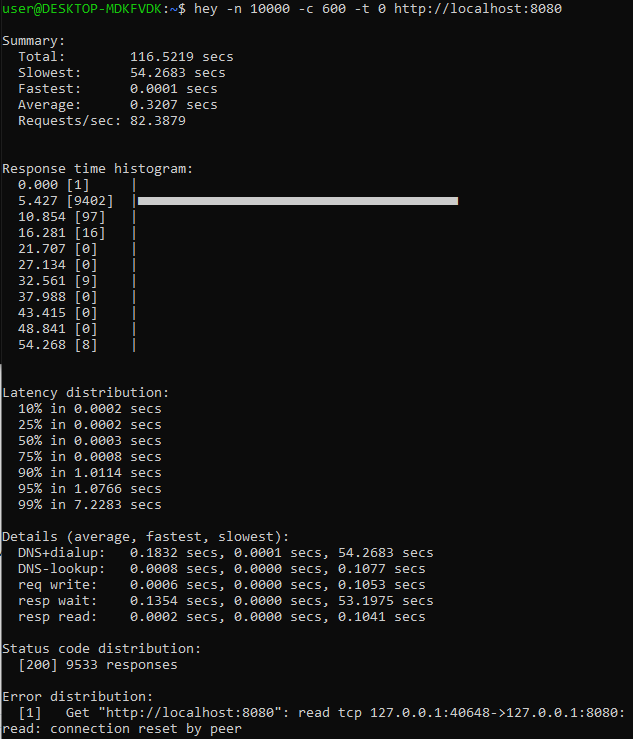


Теперь оценим максимальное количество потоков с которым может работать сервер для различных размеров стека. В контексте нашего веб-сервера каждый запрос обрабатывается в отдельном потоке. Тогда приближенной оценкой максимального количества потоков может служить максимальное количество одновременно обрабатываемых запросов (опция -c), при котором сервер не возвращает ошибки.

Начнем с 512 Кбайт. Для этого запускаем сервер следующим образом:



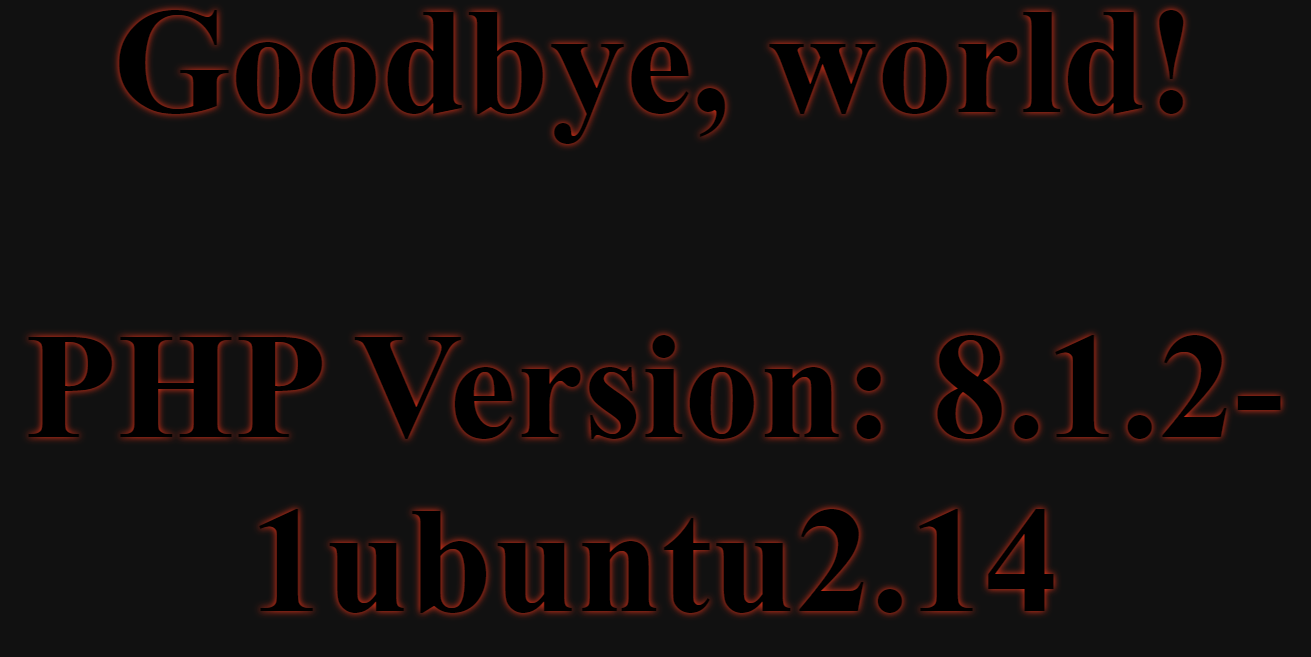
Нашли такое -c, при котором сервер достиг своего предела в обработке запросов:



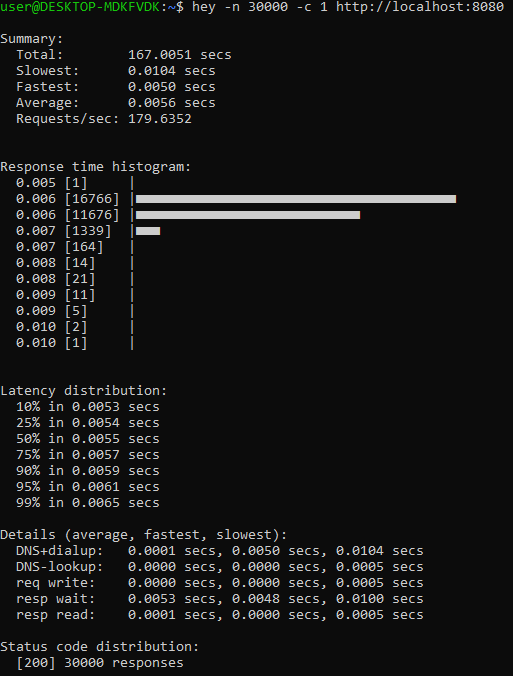
При увеличении числа одновременных запросов до 600 сервер начинает возвращать ошибки. Можно предположить, что максимальное количество потоков, с которым может работать наш сервер, составляет около 500. Это приближенная оценка.

Аналогично нашли максимальное количество потоков для остальных размеров стека. Для 1 Мбайт получилось 300 потоков, а для 2 Мбайт – 200.

Рассмотрим, как выглядит ответ от сервера, когда дополнительно возвращается версия PHP:

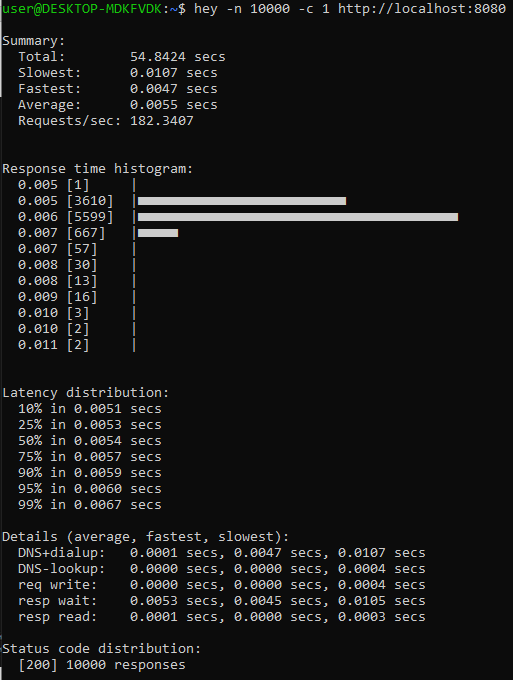


Оценим изменение производительности сервера. Протестируем с помощью hey с опциями, аналогичными самому первому тесту. Размер стека – 2 МБ. Утилита отправит 30 000 запросов без одновременных соединений:

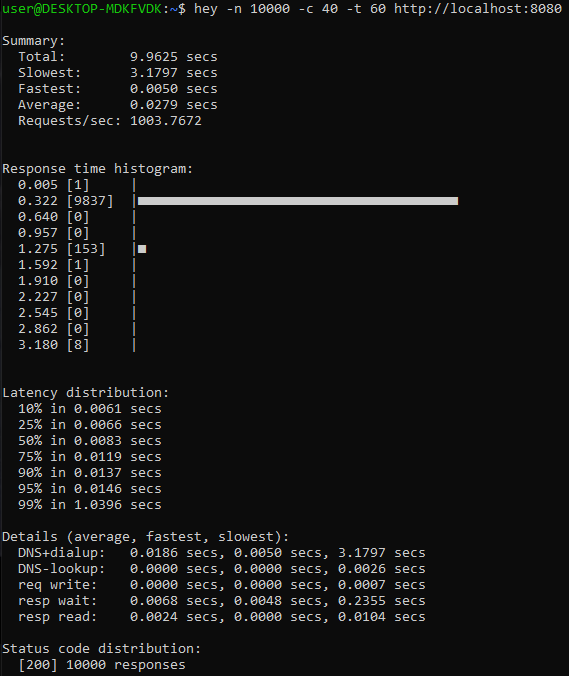


Наблюдаем значительную просадку производительности.

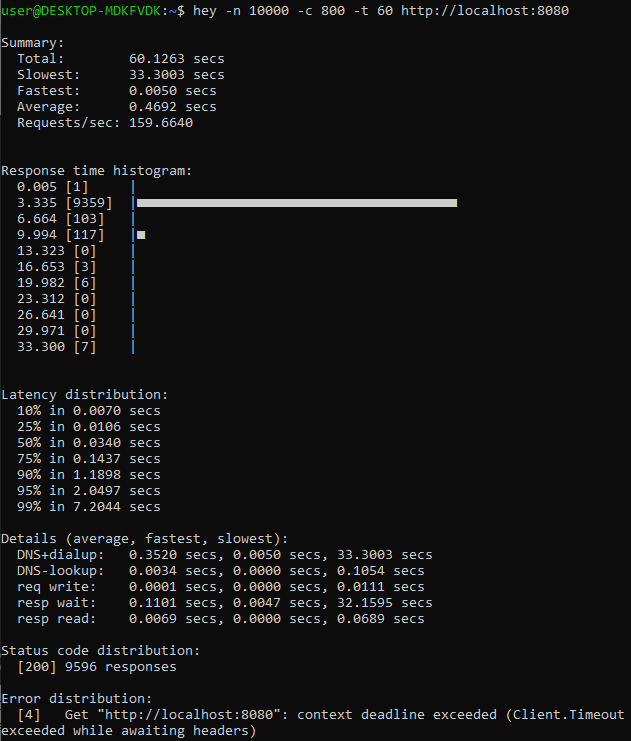
Исследуем производительность данного сервера с использованием параллельных запросов (размерность стека во всех тестах - 2 МБ). Используя меньшее -n, измерим время при последовательных запросах:



Начнем увеличивать число одновременных запросов и найдем наилучшее -c:



При числе одновременных запросов равном 40, мы получаем самое быстрое время работы утилиты hey (примерно в 6 раз быстрее, чем при последовательных запросах). Теперь найдем максимальное количество запросов, с которым может работать наш сервер. Будем считать максимальным количеством такое максимальное -c, при котором время работы утилиты ещё не превышает время работы этой же утилиты при -c равном 1 (используем опцию таймаута -t 60):



Предполагаем, что 700 параллельных запросов - максимум для нашего сервера (для всех размеров стека по умолчанию). Дальнейшее увеличение приводит к значительному увеличению времени ответа.