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

I. Цель работы

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

II. 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

III. Ход работы

1. Изучили теоретическую часть из раздела 1 настоящего пособия.

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

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ g++ Task_2.cpp -pthread -o Task_2

Task_2.cpp: In function ‘void* thread_job(void*)’:
Task_2.cpp:13:1: warning: no return statement in function returning no
n-void [-Wreturn-type]
   13 | }
      | ^
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_2
Thread is running...
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$
```

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_create()*), 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:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ g++ Task_3567.cpp -pthread -o Task_3567
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_3567 2
Thread 139692022253120 with index 0 is running...
Attribute values:
        Detach state = Joinable
        Guard size = 4096 bytes
        Stack address = 0xfffffffffff00000
        Stack size = 1048576 bytes

Thread 139692021200448 with index 1 is running...
Attribute values:
        Detach state = Joinable
        Guard size = 4096 bytes
        Stack address = 0xffffffffffe00000
        Stack size = 2097152 bytes

user@DESKTOP-MDKFVDK:~/labs_PP/lab1$
```

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:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ g++ Task_4.cpp -pthread -o Task_4
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_4 0
Execution multi thread time:  0.00015462
Execution single thread time: 0.00000007
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_4 1000
Execution multi thread time:  0.00015728
Execution single thread time: 0.00000414
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_4 100000
Execution multi thread time:  0.00028948
Execution single thread time: 0.00031761
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_4 100000
Execution multi thread time:  0.00027009
Execution single thread time: 0.00031555
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_4 1000000
Execution multi thread time:  0.00162382
Execution single thread time: 0.00296084
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_4 10000000
Execution multi thread time:  0.01587216
Execution single thread time: 0.02811087
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$
```

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):

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ g++ Task_8.cpp -pthread -o Task_8
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_8 5 5 2 add
1 2 3 4 5
3 4 5 6 7
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_8 5 10 5 mul
1 2 3 4 5
5 10 15 20 25
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_8 10 5 7 sub
1 2 3 4 5 6 7 8 9 10
-6 -5 -4 -3 -2 -1 0 1 2 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Task_8 11 6 1 add
1 2 3 4 5 6 7 8 9 10 11
2 3 4 5 6 7 8 9 10 11 12
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$
```

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;
```

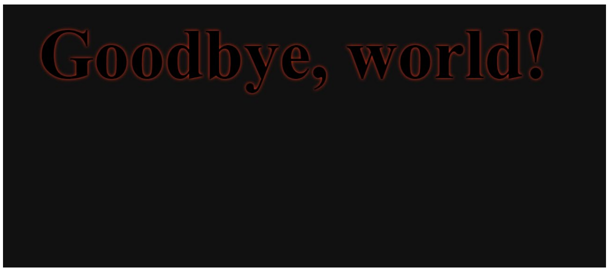
```
	cout << endl << "Server listening on port 8080..." << endl;

	// 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:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Server
Please choose the execution method for the program:
        0 - Normal execution
        1 - PHP execution
Enter 0 or 1: 0

Server listening on port 8080...
```

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:

```
user@DESKTOP-MDKFVDK:~$ hey -n 30000 -c 1 http://localhost:8080

Summary:
  Total:        5.6480 secs
  Slowest:      0.0009 secs
  Fastest:      0.0001 secs
  Average:      0.0002 secs
  Requests/sec: 5311.6440


Response time histogram:
  0.000 [1]     |
  0.000 [11538] |■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.000 [17883] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.000 [318]   |■
  0.000 [103]   |
  0.000 [86]    |
  0.001 [45]    |
  0.001 [18]    |
  0.001 [3]     |
  0.001 [3]     |
  0.001 [2]     |


Latency distribution:
  10% in 0.0002 secs
  25% in 0.0002 secs
  50% in 0.0002 secs
  75% in 0.0002 secs
  90% in 0.0002 secs
  95% in 0.0002 secs
  99% in 0.0003 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0001 secs, 0.0001 secs, 0.0009 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0003 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0003 secs
  resp wait:    0.0001 secs, 0.0000 secs, 0.0007 secs
  resp read:    0.0000 secs, 0.0000 secs, 0.0004 secs

Status code distribution:
  [200] 30000 responses
```

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

```
user@DESKTOP-MDKFVDK:~$ hey -n 30000 -c 12 http://localhost:8080

Summary:
  Total:        1.5800 secs
  Slowest:      1.0823 secs
  Fastest:      0.0001 secs
  Average:      0.0004 secs
  Requests/sec: 18987.7585


Response time histogram:
  0.000 [1]     |
  0.108 [29993] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.217 [0]     |
  0.325 [0]     |
  0.433 [0]     |
  0.541 [0]     |
  0.649 [0]     |
  0.758 [0]     |
  0.866 [0]     |
  0.974 [0]     |
  1.082 [6]     |


Latency distribution:
  10% in 0.0001 secs
  25% in 0.0002 secs
  50% in 0.0002 secs
  75% in 0.0002 secs
  90% in 0.0002 secs
  95% in 0.0003 secs
  99% in 0.0006 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0003 secs, 0.0001 secs, 1.0823 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0011 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0015 secs
  resp wait:    0.0001 secs, 0.0000 secs, 0.0026 secs
  resp read:    0.0000 secs, 0.0000 secs, 0.0028 secs

Status code distribution:
  [200] 30000 responses
```

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:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab1$ ./Server 0.5
Please choose the execution method for the program:
        0 - Normal execution
        1 - PHP execution
Enter 0 or 1: 0

Server listening on port 8080...
```

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

```
user@DESKTOP-MDKFVDK:~$ hey -n 10000 -c 600 -t 0 http://localhost:8080

Summary:
  Total:        116.5219 secs
  Slowest:      54.2683 secs
  Fastest:      0.0001 secs
  Average:      0.3207 secs
  Requests/sec: 82.3879


Response time histogram:
  0.000 [1]     |
  5.427 [9402]  |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  10.854 [97]   |
  16.281 [16]   |
  21.707 [0]    |
  27.134 [0]    |
  32.561 [9]    |
  37.988 [0]    |
  43.415 [0]    |
  48.841 [0]    |
  54.268 [8]    |


Latency distribution:
  10% in 0.0002 secs
  25% in 0.0002 secs
  50% in 0.0003 secs
  75% in 0.0008 secs
  90% in 1.0114 secs
  95% in 1.0766 secs
  99% in 7.2283 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.1832 secs, 0.0001 secs, 54.2683 secs
  DNS-lookup:   0.0008 secs, 0.0000 secs, 0.1077 secs
  req write:    0.0006 secs, 0.0000 secs, 0.1053 secs
  resp wait:    0.1354 secs, 0.0000 secs, 53.1975 secs
  resp read:    0.0002 secs, 0.0000 secs, 0.1041 secs

Status code distribution:
  [200] 9533 responses

Error distribution:
  [1]   Get "http://localhost:8080": read tcp 127.0.0.1:40648->127.0.0.1:8080:
read: connection reset by peer
```

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:

Goodbye, world!

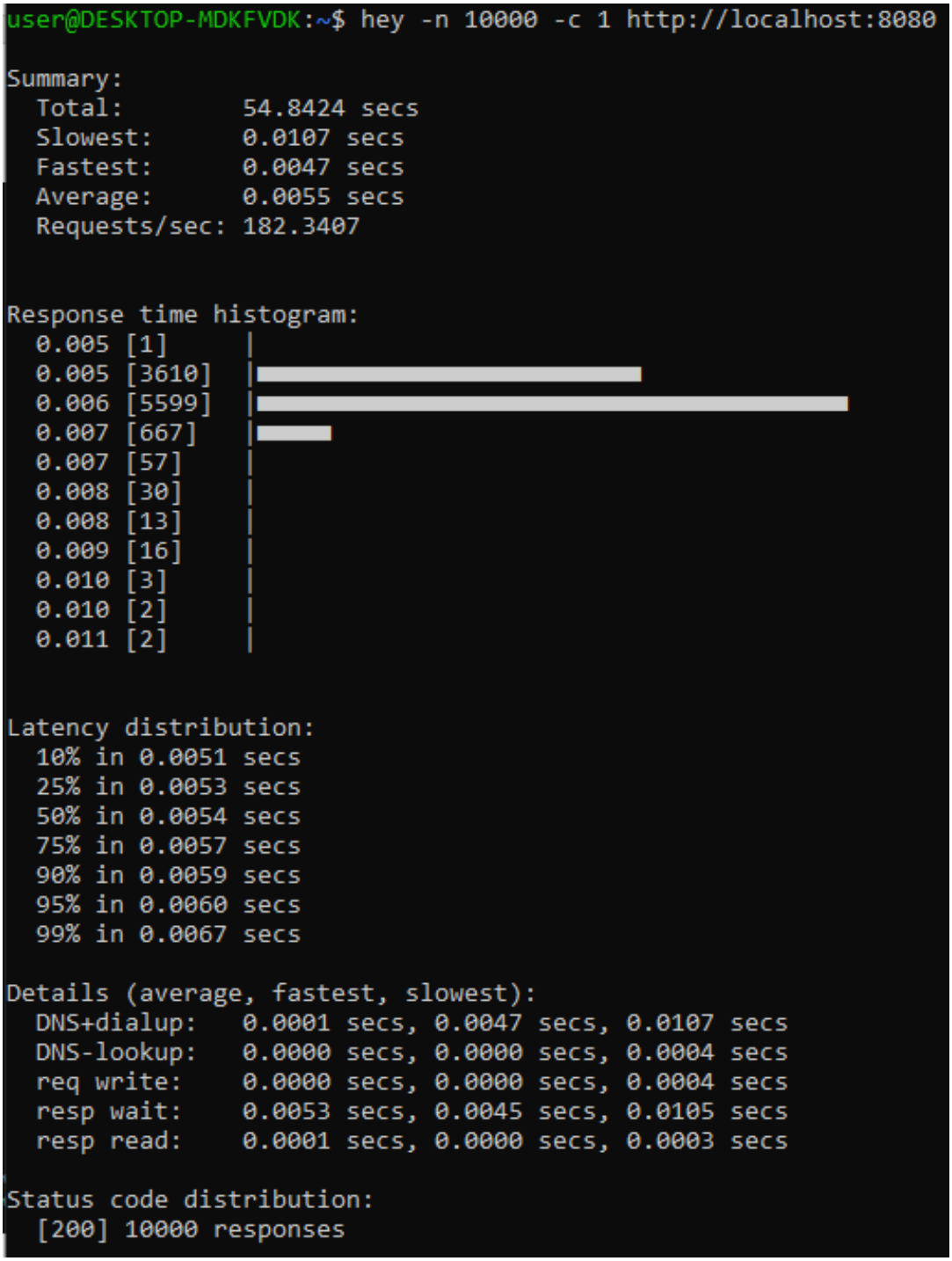
PHP Version: 8.1.2-1ubuntu2.14

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:

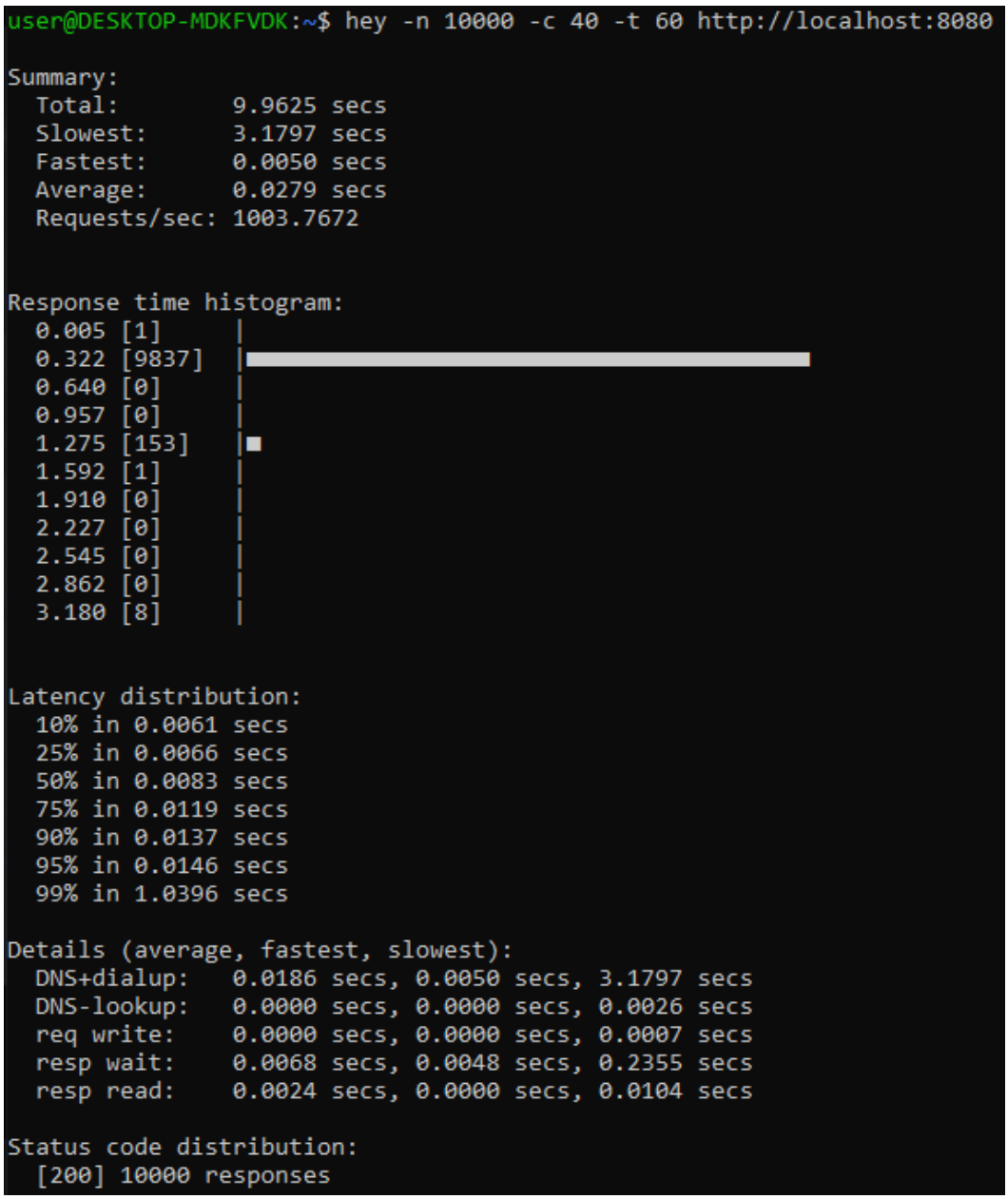
```
user@DESKTOP-MDKFVDK:~$ hey -n 30000 -c 1 http://localhost:8080

Summary:
  Total:        167.0051 secs
  Slowest:      0.0104 secs
  Fastest:      0.0050 secs
  Average:      0.0056 secs
  Requests/sec: 179.6352


Response time histogram:
  0.005 [1]     |
  0.006 [16766] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.006 [11676] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.007 [1339]  |■■■
  0.007 [164]   |
  0.008 [14]    |
  0.008 [21]    |
  0.009 [11]    |
  0.009 [5]     |
  0.010 [2]     |
  0.010 [1]     |


Latency distribution:
  10% in 0.0053 secs
  25% in 0.0054 secs
  50% in 0.0055 secs
  75% in 0.0057 secs
  90% in 0.0059 secs
  95% in 0.0061 secs
  99% in 0.0065 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0001 secs, 0.0050 secs, 0.0104 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0005 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0005 secs
  resp wait:    0.0053 secs, 0.0048 secs, 0.0100 secs
  resp read:    0.0001 secs, 0.0000 secs, 0.0005 secs

Status code distribution:
  [200] 30000 responses
```

Наблюдаем значительную просадку производительности.

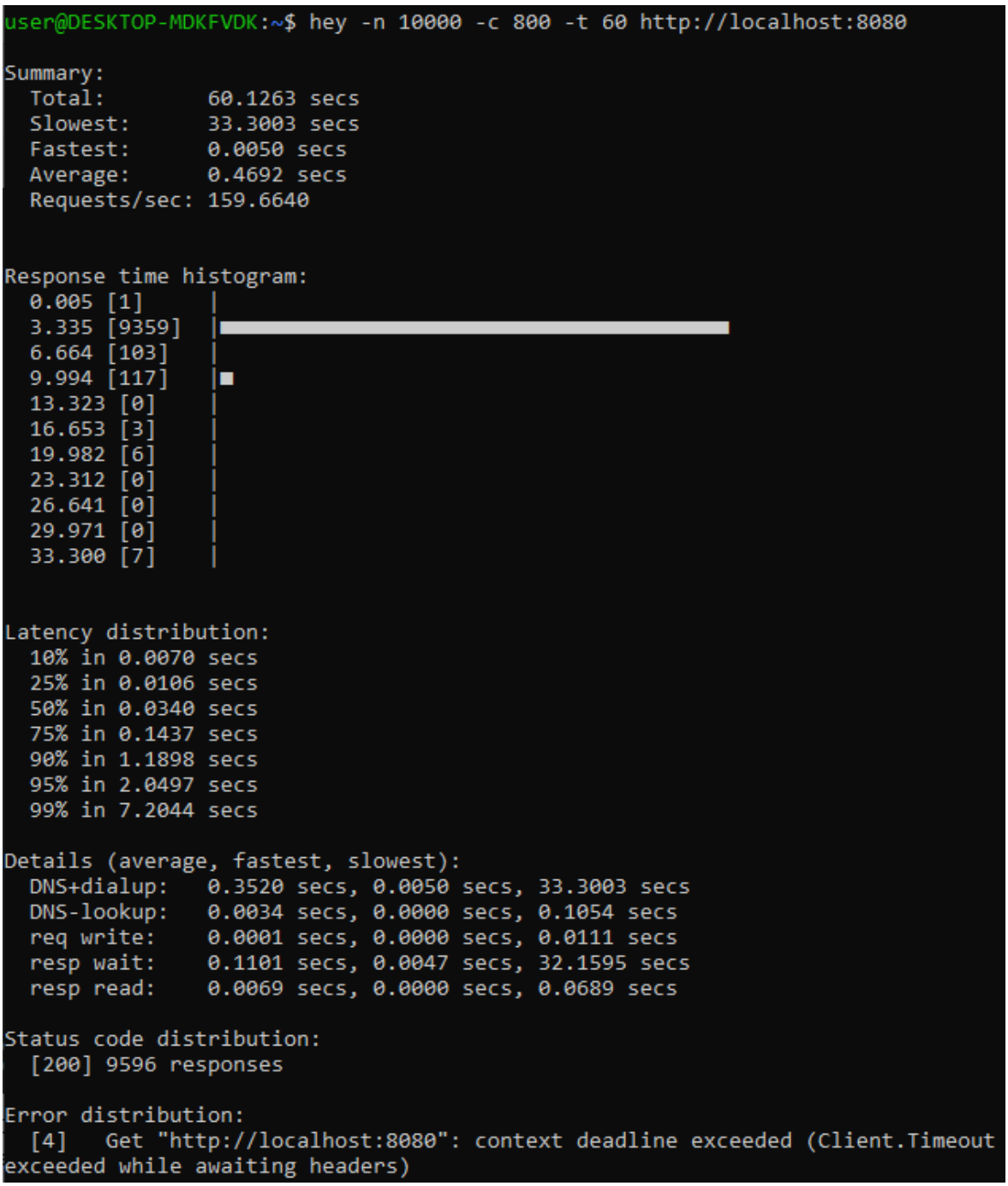
Исследуем производительность данного сервера с использованием параллельных запросов (размерность стека во всех тестах - 2 МБ). Используя меньшее -n, измерим время при последовательных запросах:



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.