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 № 3 по курсу "Анализ алгоритмов"

Тема Многопоточное программирование

Студент Калашков П. А.

Группа ИУ7-56Б

Оценка (баллы)

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

Москва — 2022 г.

Содержание

Введение

По мере развития вычислительных систем программисты столкнулись с необходимостью производить параллельную обработку данных для улучшения отзывчивости системы, ускорения производимых вычислений и рационального использования вычислитлельных мощностей. Благодаря развитию процессоров стало возможным использовать один процессор для выполнения нескольких параллельных операций, что дало начало термину “многопоточность“.

Цель работы: изучение параллельного выполнения операций на примере сервера раздачи статической информациий Для достижения поставленной цели необходимо выполнить следующие задачи:

1) изучить основы распараллеливания вычислений;

2) разработать и реализовать программное обеспечение, позволяющее раздавать статическую информацию на локальном сервере согласно патерну thread pool;

3) сравить и проанализировать по времени обработки установленного количества поданных запросов с использованием многопоточности и без неё;

4) описать и обосновать полученные результаты в виде отчёта о выполненной лабораторной работе, выполненном как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В этом разделе будет представлена информация о многопоточности, а также теоретически описан паттерн thread pool, реализация которого будет распараллелена в данной лабораторной работе.

1.1 Многопоточность

Многопоточность [1] (англ. *multithreading*)– это способность центрального процессора (ЦП) обеспечивать одновременное выполнение нескольких потоков в рамках использования ресурсов одного процессора. Поток – последовательность инструкций, которые могут исполняться параллельно с другими потоками одного и того же породившего их процесса.

Процессом [2] называют программу в ходе своего выполнения. Таким образом, когда запускается программа или приложение, создаётся процесс. Один процесс может состоять из одного или больше потоков. Таким образом, поток является сегментом процесса, сущностью, которая выполняет задачи, стоящие перед исполняемым приложением. Процесс завершается тогда, когда все его потоки заканчивают работу. Каждый поток в операционной системе является задачей, которую должен выполнить процессор. Сейчас большинство процессоров умеют выполнять несколько задач на одном ядре, создавая дополнительные, виртуальные ядра, или же имеют несколько физических ядер. Такие процессоры нызываются многоядерными.

При написании программы, использующей несколько потоков, следует учесть, что при последовательном запуске потоков и передаче управления исполняемому потоку не получится раскрыть весь потенциал многопоточности, т.к. большинство потоков будут существовать без дела. Необходимо создавать потоки для независимых, отнимающих много времени функций, и выполнять их параллельно, тем самым сокращая общее время выполнения процесса.

Одна из проблем, встающих при использовании потоков, является проблема совместного доступа к информации. Фундаментальным ограничением является запрет на запись из двух и более потоков в одну ячейку памяти

одновременно.

Из того следует, что необходим примитив синхронизации обращения к данным - так называемый мьютекс (англ. *mutex - mutual exclusion*). Он может быть захвачен для работы в монопольном режиме или освобождён. Так, если 2 потока одновременно пытаются захватить мьютекс, успевает только один, а другой будет ждать освобождения.

Набор инструкций, выполняемых между захватом и освобождением мьютекса, называется *критической секцией*. Поскольку в то время, пока мьютекс захвачен, остальные потоки, требующие выполнения критической секции, ждут освобождения мьютекса, требуется разрабатывать программное обеспечение таким образом, чтобы критическая секция был минимальной.

1.2 Паттерн thread pool

Thread pool [3] – архитектурный паттерн, предназначенный для достижения параллельности выполнения программы. Основная идея данного паттерна заключается в том, чтобы создавать набор (пул) потоков, из которого в случае появления задачи, которую можно выполнить параллельно с другими потоками, берётся поток. Задача выполняется в рамках забранного потока и после завершения освобождённый поток возвращается в пул.

Согласно поставленной задаче, реализации сервера для раздачи статической информации, можно сделать вывод, что задачами, которые можно выполнять параллельно, является обработка запроса. Поступившие запросы необходимо синхронно записывать в очередь запросов, после чего распределять задачи по имеющимся потокам.

Вывод

В данной работе необходимо реализовать сервер раздачи статической информации с использованием архитектурного паттерна thread pool, а также провести сравнение времени обработки установленного количества запросов при использовании различного числа потоков. Требуется, чтобы сервер умел

отдавать информацию различного типа и размера, в том числе полноценные статические страницы, директории, вложенные файлы, а также пустые запросы.

2 Конструкторская часть

В данном разделе будут рассмотрены схема алгоритмов, реализующих сервер раздачи статической информации согласно паттерну thread pool с распараллеливанием, а также без него.

2.1 Требования к ПО

Ряд требований к программе:

- сервер должен работать на локальном доменном имени на порту 8080;
- обращение к серверу происходит посредством браузера, запросы HTTP1.1;
- в ответ должен отдаваться статический файл, указанный в предыдущем пункте.

2.2 Разработка алгоритмов

На рисунках 2.1, 2.2 представлены схемы алгоритмов помещения запроса в очередь, а также обработки запроса в потоке. Заметим, что реализация сервера согласно данным схемам будет одинаковой как в случае исполльзовании нескольких потоков, так и в случае отсутствия параллельности, в силу того, что даже непараллельная реализация является реализацией с использованием одного потока.

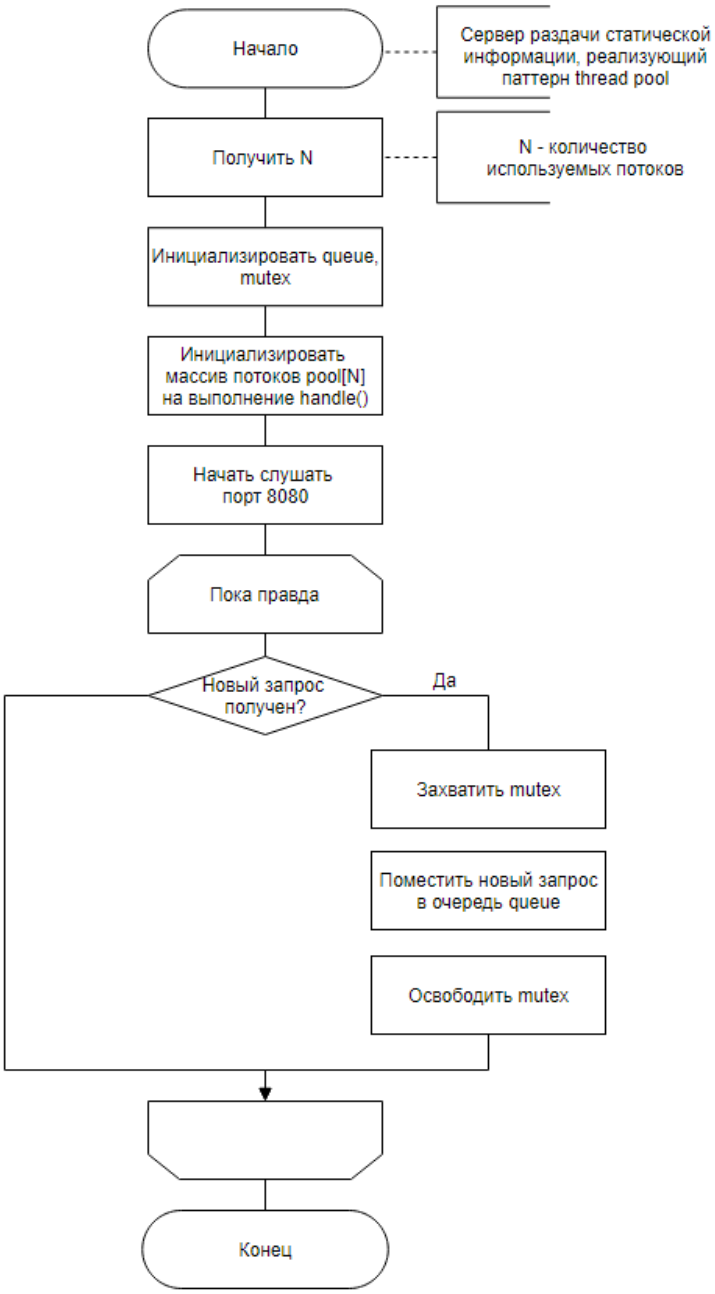

Рисунок 2.1 – Схема алгоритма сервера помещения запроса в очередь

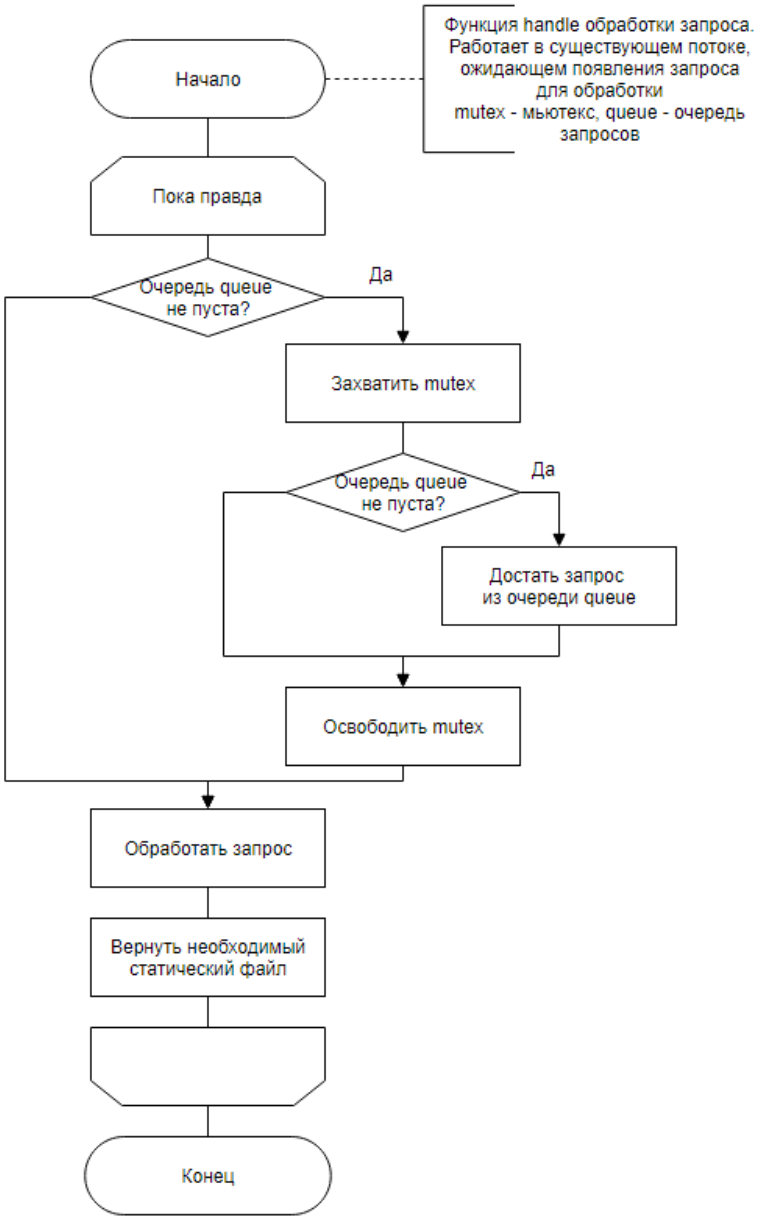

Рисунок 2.2 – Схема алгоритма обработки запроса в потоке

Вывод

Были разработаны схемы всех алгоритмов помещения запроса в очередь, а также обработки и получения необходимого запроса из очередь в потоке. Критическая секция была снижена настолько, насколько было возможно, что позволит наиболее эффективно использовать многопоточность.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги реализаций рассматриваемых сортировок.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования $C++$[4]. В текущей лабораторной работе требуется замерить процессорное время работы выполняемой программы и визуализировать результаты при помощи графиков. Инструменты для этого присутствуют в выбранном языке программирования.

Построение графиков и функциональное тестирование было осуществлено при помощи языка программирования $Python$[5], нагрузочное тестирование и замеры времени при помощи утилиты wrk[6].

3.2 Сведения о модулях программы

Программа состоит из шести модулей:

1) $main$ – файл, содержащий точку входа;

2) $Server$ – модуль, содержащий реализацию сервера;

3) $utils$ – модуль, содержащий служебные алгоритмы;

4) $constants.hpp$ – файл, содержаший константы программы;

5) $responses.hpp$ – файл, содержащий шаблоны ответов.

3.3 Реализации алгоритмов

В листингах 3.1, 3.2,3.3 представлены реализации алгоритмов создания пула потоков, помещения запросов в очередь и обработки запроса в потоке

Листинг 3.1 – Заголовочный файл реализации сервера

```
1  class Server {
2      public:
3      Server() = delete;
4
5      explicit Server(size_t poolSize = POOL_SIZE, int port = PORT,
           int countConn = CONN_COUNT)
6      :poolSize(poolSize), serverPort(port),
           queueConnections(CONN_COUNT) {
7          this->serverSocket = 0, this->clientSocket =  0;
8          memset(&this->serverAddress, 0, sizeof(struct sockaddr_in));
9          memset(&this->clientAddress, 0, sizeof(struct sockaddr_in));
10         this->threadPool.resize(this->poolSize);
11         for(size_t i = 0; i < this->poolSize; i++) {
12             this->threadPool[i] =
                   std::thread([this](){this->handleRequest();});
13         }
14     };
15
16     int init();
17     void run();
18
19     private:
20     int serverSocket;
21     int clientSocket;
22     struct sockaddr_in serverAddress;
23     struct sockaddr_in clientAddress;
24     int serverPort;
25
26     std::vector<std::thread> threadPool;
27     size_t poolSize;
28     std::queue<int> requestQueue;
29
30     int queueConnections;
31     std::mutex queueMutex;
32     std::condition_variable cv;
33
34     void handleRequest();
35     int loadConfig();
36 };
```

Листинг 3.2 – Запуск сервера и алгоритм помещения запросов в очередь

```
void Server::run() {
    while (true) {
        socklen_t addr_size = sizeof(this->clientAddress);
        this->clientSocket = accept(serverSocket, (struct sockaddr
            *) &this->clientAddress, &addr_size);
        char peerIP[INET_ADDRSTRLEN] = {0};
        if (inet_ntop(AF_INET, &this->clientAddress.sin_addr,
            peerIP, sizeof(peerIP))) {
            std::cout << "Accepted connection with " << peerIP <<
                "\n";
        } else {
            return;
        }
        this->queueMutex.lock();
        this->requestQueue.push(clientSocket);
        this->cv.notify_one();
        this->queueMutex.unlock();
    }
}
```

Листинг 3.3 – Алгоритм обработки запроса в потоке

```
void Server::handleRequest() {
    std::unique_lock<std::mutex> lock(this->queueMutex,
        std::defer_lock);
    int client_socket = -1;
    while (true) {
        lock.lock();
        this->cv.wait(lock, [this]() { return
            !this->requestQueue.empty(); });
        client_socket = this->requestQueue.front();
        this->requestQueue.pop();
        lock.unlock();
        char req[2 * REQ_SIZE];
        recv(client_socket, req, sizeof(req), 0);
        std::string reply = handle(req);
        send(client_socket, reply.c_str(), reply.size(), 0);
        close(client_socket);
    }
}
```

3.4 Функциональные тесты

В таблице 3.1 приведены запросы для написанного сервера раздачи статической информации. Тесты для всех запросов пройдены *успешно*.

Таблица 3.1 – Функциональные тесты

Содержимое URL	Ожидаемый результат
Пусто	Ответа нет
Заголовок HTTP1.1	Ответа нет
/test	Server
/test/dirToTest	/test/dirToTest/index.html
/test/deep	Статус 403
/test/blablabla.txt	Статус 403
/test/deep/deeper/deepest/deep.txt	bingo, you found it
/test/dirToTest/page.html/	Статус 404
/test/dirToTest/page.html?arg=val	Статус 404
/test/s%20p%20a%20c%20e.txt	s p a c e.txt
/test/dirToTest/%61%2e%68%74%6d%6c	page.html
/test/wikipedia_russia.html	Страница Википедии
/test/../../../../../../../../../../../../etc/passwd	Статус 403
/test/text..txt	text..txt
/test/dirToTest/page.html	page.html
/test/css.css	css.css
/test/js.js	js.js
/test/jpg.jpg	jpg.jpg
/test/jpeg.jpeg	jpeg.jpeg
/test/png.png	png.png
/test/gif.gif	gif.gif
/test/swf.swf	swf.swf

Вывод

В данном разделе были выбраны средства реализации, представлены листинги реализаций алгоритмов создания пула потоков, помещения запросов в очередь и обработки запроса в потоке, а также рассмотрены запросы для функционального тестирования, которое для всех запросов проходит успешно.

4 Исследовательская часть

В данном разделе будут приведён пример работы программы, а также проведён сравнительный анализ многопоточной реализации и однопоточной

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

- операционная система: Mac OS Monterey Версия 12.5.1 (21G83) [7] x86_64;
- память: 16 GB;
- процессор: 2,7 GHz 4-ядерный процессор Intel Core i7 [8].

При тестировании ноутбук был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

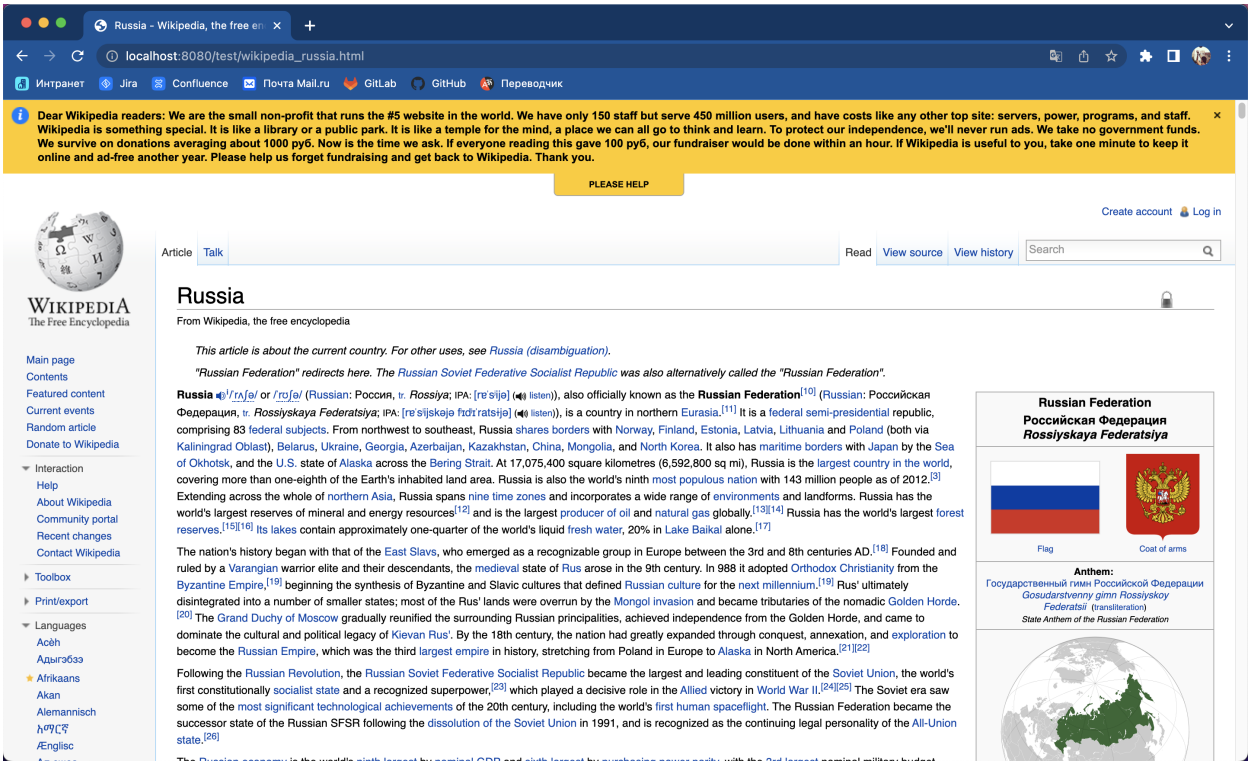

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Для проведения нагрузочного тестирования и замера времени обработки запросов используется утилита wrk. Нагрузочное тестирование будет производиться в течение 30 секунд с 12 потоков и с указанным числом соединений, результаты (количество обработанных запросов в секунду – requests per second, rps к количеству открытых соединений)

Результаты тестирования приведены в таблице 4.1.

Таблица 4.1 – Результаты нагрузочного тестирования

Количество соединений, шт.	1 поток, rps	8 потоков, rps	16 потоков, rps
12	537	544	550
40	291	434	546
80	189	387	543
120	147	360	533
160	117	319	514
200	86	220	506

На рисунке 4.2 приведены графические результаты нагрузочного тестирования.

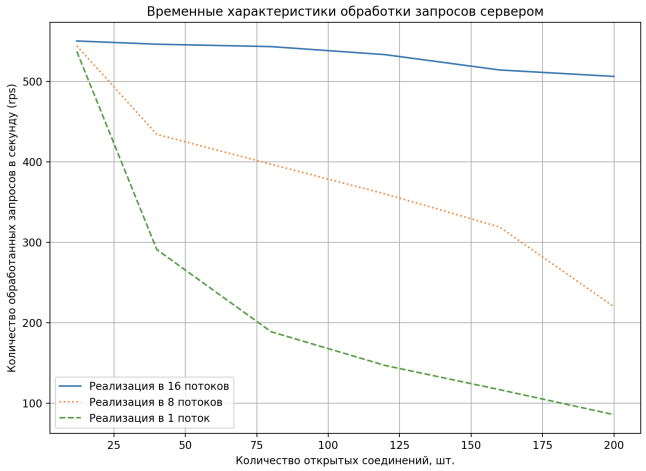

Рисунок 4.2 – Результаты нагрузочного тестирования

Таким образом, при увеличении до 200 открытых соединений в течении 30 секунд реализация с использованием 16 потоков одновременно оказалась бо-

лее чем в 6 раз эффективнее реализации с одним потоком, а с использованием 8 потоков - почти в 3 раза.

Вывод

Исходя из полученных результатов, при увеличении нагрузки на сервер (например, увеличения открытых соединений) реализация с большим количеством потоков показала наилучший результат – так, реализация с использованием 16 потоков одновременно оказалась более чем в 6 раз эффективнее реализации с одним потоком. В целом, использование многопоточности показало значительный прирост количества обработанных запросов в секунду, в частности при количестве открытых запросов больше, чем 20.

Теоретические результаты замеров и полученные практически результаты совпадают.

Заключение

Цель, которая была поставлена в начале лабораторной работы была достигнута, а также в ходе выполнения лабораторной работы были решены следующие задачи:

1) были изучены основы распараллеливания вычислений;

2) было разработатано и реализовано программное обеспечение, позволяющее раздавать статическую информацию на локальном сервере согласно паттерну thread pool;

3) были проведены сравнение и анализ по времени обработки установленного количества поданных запросов с использованием многопоточности и без неё;

4) описать и обосновать полученные результаты в виде отчёта о выполненной лабораторной работе, выполненном как расчётно-пояснительная записка к работе.

5) подготовлен отчёт о лабораторной работе, представленный как расчётно-пояснительная записка к работе.

Исходя из полученных результатов, при увеличении нагрузки на сервер (например, увеличения открытых соединений) реализация с большим количеством потоков показала наилучший результат – так, реализация с использованием 16 потоков одновременно оказалась более чем в 6 раз эффективнее реализации с одним потоком. В целом, использование многопоточности показало значительный прирост количества обработанных запросов в секунду, в частности при количестве открытых запросов больше, чем 20.

Теоретические результаты замеров и полученные практически результаты совпадают.

Список литературы

[1] Stoltzfus J. Multithreading. – Techopedia - Janalta Interactive Inc., 2022. URL: https://www.techopedia.com/definition/24297/multithreading-computer-architecture.

[2] У. Ричард Стивенс Стивен А. Раго. UNIX. Профессиональное программирование. 3-е издание. – СПб.: Питер., 2018. С. – 994.

[3] Christudas B. Query by Slice, Parallel Execute, and Join: A Thread Pool Pattern in Java. – Infosys Technologies, 2008. URL: https://web.archive.org/web/20080207124322/http://today.java.net/pub/a/today/2008/01/31/query-by-slice-parallel-execute-join-thread-pool-pattern.html.

[4] Программирование на C/C++ [Электронный ресурс]. Режим доступа: http://www.c-cpp.ru/ (дата обращения: 11.10.2021).

[5] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 11.10.2022).

[6] Утилита wrk [Электронный ресурс]. Режим доступа: https://linuxcommandlibrary.com/man/wrk (дата обращения: 11.10.2022).

[7] macOS Monterey [Электронный ресурс]. Режим доступа: https://www.apple.com/macos/monterey/ (дата обращения: 11.10.2022).

[8] Процессор Intel® Core™ i7 [Электронный ресурс]. Режим доступа: https://www.intel.com/processors/core/i7/docs (дата обращения: 17.09.2022).