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»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Разработка статического сервера»

Студент ИУ7-74Б _____ А. В. Золотухин
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ _____
(Подпись, дата) (И.О.Фамилия)

2023 г.

РЕФЕРАТ

Курсовая работа представляет собой реализацию статического веб-сервера с использованием мультиплексора select.

Ключевые слова: статический веб-сервер, мультиплексор, select, C.

Расчетно-пояснительная записка к курсовой работе содержит 21 страницы, 3 иллюстраций, 3 таблицы, 6 источников.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Веб-сервером [1] называется не только оборудование, но и обслуживающие веб-сервер программы. Также этим словом обозначается и то, и другое в совокупности.

Оборудование для веб-сервера представляет собой хранилище файлов сайта. На нем хранятся как отдельные страницы и файлы стилей, так и мультимедийные файлы – аудио, видео, графика и др. С сервера контент попадает на компьютер, с которого был отправлен запрос, и выводится в наглядном виде через браузер. Программная составляющая веб-сервера позволяет осуществлять управление размещенными на нем данными, обеспечивает доступ пользователей. Минимально для этого требуется HTTP-сервер, то есть программа, которая может распознавать URL-адреса и работает на протоколе HTTP, который необходим для доступа к веб-странице.

Веб-серверы для публикации сайтов делятся на статические и динамические. Статические веб-серверы — это «железо» с установленным на нем ПО для HTTP, которое направляет размещенные файлы в браузер в неизменном виде.

В динамических веб-серверах на статические веб-сервера устанавливается дополнительное программное обеспечение, чаще всего сервера приложения и базы данных. В таких серверах исходные файлы изменяются перед отправкой по HTTP.

Цель курсовой работы — разработать статический веб-сервер.

Для достижения поставленной цели необходимо решить следующие задачи:

— провести анализ предметной области и формализовать задачу;

— спроектировать структуру программного обеспечения;

— реализовать программное обеспечение, которое будет обслуживать контент, хранящийся во вторичной памяти;

— провести нагрузочное тестирование и сравнить с распространёнными аналогами.

1 АНАЛИТИЧЕСКАЯ ЧАСТЬ

В данном разделе будут формализованы задача и данные. Будут описаны теоретические сведения, которые нужны для решения поставленной задачи.

1.1 Протокол HTTP

HTTP [2] — это протокол, позволяющий получать различные ресурсы, например HTML-документы. Протокол HTTP лежит в основе обмена данными в Интернете. HTTP является протоколом клиент-серверного взаимодействия, что означает инициирование запросов к серверу самим получателем, обычно веб-браузером. Полученный итоговый документ будет (может) состоять из различных поддокументов, являющихся частью итогового документа: например, из отдельно полученного текста, описания структуры документа, изображений, видео-файлов, скриптов и многого другого.

HTTP — это клиент-серверный протокол, то есть запросы отправляются какой-то одной стороной — участником обмена (user-agent) (либо прокси вместо него). Чаще всего в качестве участника выступает веб-браузер, но им может быть кто угодно, например, робот, путешествующий по Сети для пополнения и обновления данных индексации веб-страниц для поисковых систем.

Структура HTTP-сообщения всегда одинакова:

1. Стартовая строка, в которой определяется адрес, по которому отправляется запрос, и тип сообщения. Указывается метод, который определяет действия при получении этого сообщения. Это может быть чтение данных, их отправка, изменение или удаление.

2. Заголовки (Headers), в которых прописаны определённые параметры сообщения. Например, может быть напрямую задан язык.

3. Тело запроса (Request Body), текст сообщения — данные, которые передаются. Например, файлы, отправляемые на сервер.

1.2 Сокеты

Сокеты — 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 сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Каждый процесс может создать слушающий сокет и привязать его к какому-нибудь порту операционной системы (в UNIX непривилегированные процессы не могут использовать порты меньше 1024). Слушающий процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. При этом сохраняется возможность проверить наличие соединений на данный момент, установить тайм-аут для операции и т.д.

Каждый сокет имеет свой адрес. ОС семейства UNIX могут поддерживать много типов адресов, но обязательными являются INET-адрес и UNIX-адрес. Если привязать сокет к UNIX-адресу, то будет создан специальный файл (файл сокета) по заданному пути, через который смогут сообщаться любые локальные процессы путём чтения/записи из него. Сокеты типа INET доступны из сети и требуют выделения номера порта.

Обычно клиент явно подсоединяется к слушателю, после чего любое чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером.

В таблице 1.1 представлены основные функции для работы с сокетами.

Таблица 1.1 – Основные функции для работы с сокетами

Общие	
Socket	Создать новый сокет и вернуть файловый дескриптор
Send	Отправить данные по сети
Receive	Получить данные из сети
Close	Закрыть соединение
Серверные	
Bind	Связать сокет с IP-адресом и портом
Listen	Объявить о желании принимать соединения. Слушает порт и ждет когда будет установлено соединение
Accept	Принять запрос на установку соединения
Клиентские	
Connect	Установить соединение

1.3 Мультиплексирование ввода/вывода

При мультиплексировании ввода/вывода [3] мы обращаемся к одному из доступных в ОС системному вызову (мультиплексору), например select, poll, pselect, dev/poll, epoll и на нем блокируемся вместо того, чтобы блокироваться на фактическом I/O вызове. Схематично процесс мультиплексирования представлен на рисунке 1.1.

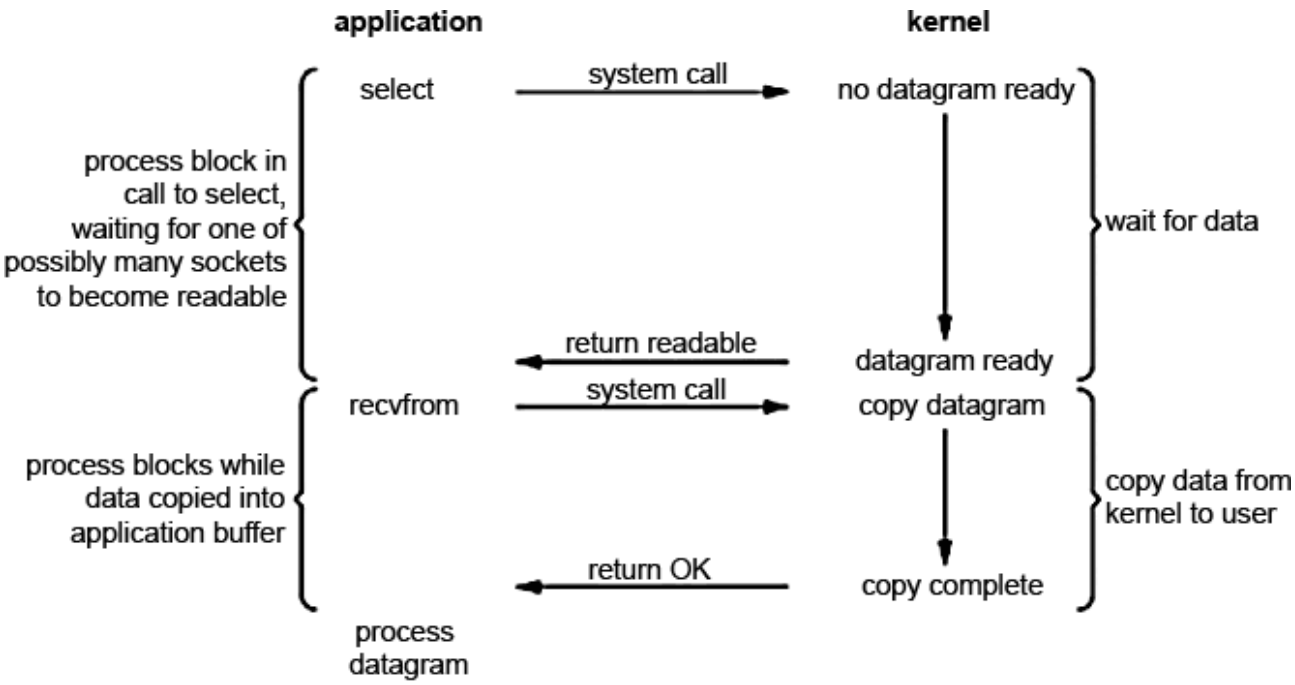

Рисунок 1.1 – Схема мультиплексированного ввода/вывода

Приложение блокируется при вызове select'а, ожидая, когда сокет станет доступным для чтения. Затем ядро возвращает нам статус readable и можно получать данные помощью recvfrom. В отличии от блокирующего метода, мультиплексор позволяет ожидать данные не от одного, а от нескольких файловых дескрипторов. Схема работы системного вызова select изображена на рисунке 1.2.

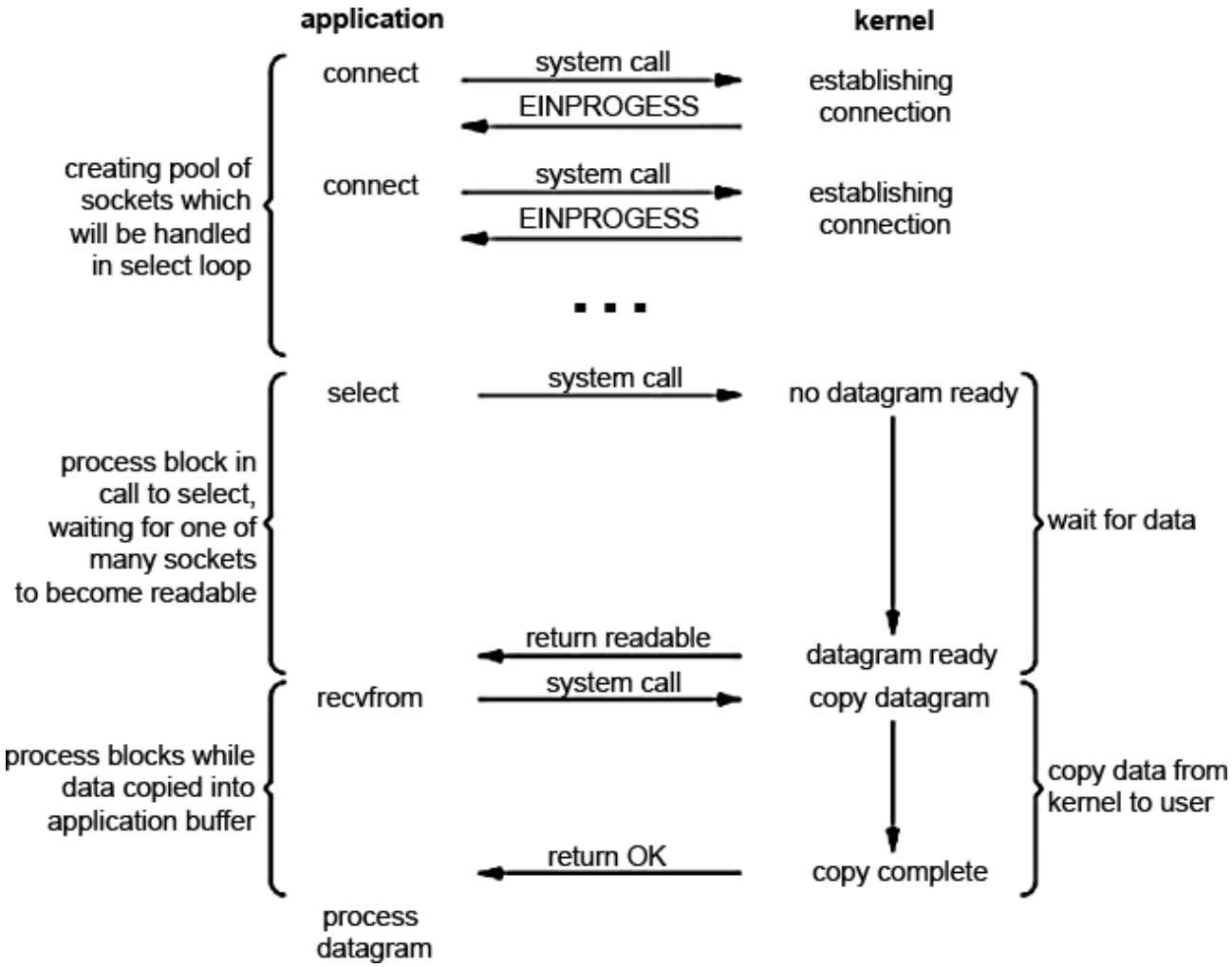

Рисунок 1.2 – Схема работы системного вызова select

1.4 Способ параллелизации обработки запросов

Thread Pool поддерживает несколько потоков, ожидающих распределения задач для параллельного выполнения. Когда в систему поступает новая задача, она помещается в очередь, и один из доступных потоков в пуле забирает эту задачу на выполнение. После завершения задачи поток возвращается обратно в пул и становится доступным для выполнения новых задач. Пул потоков повышает производительность и позволяет избежать задержек в выполнении из-за частого создания и уничтожения потоков для кратковременных задач.

Prefork — это способ параллелизации, при котором родительский процесс создаёт дочерние процессы для выполнения задач. Такой подход позволяет обрабатывать каждый запрос изолированно от остальных запросов, что

повышает надежность, так как сбои в одном процессе не влияют на остальные. Однако создание процессов требует больше дополнительных ресурсов, чем использование пула потоков.

2 КОНСТРУКТОРСКАЯ ЧАСТЬ

В данном разделе будут представлены схемы алгоритмов работы веб-сервера.

2.1 Схема алгоритмов работы веб-сервера

На рисунке 2.1 представлена схема алгоритма работы веб-сервера, который с помощью сокетов и мультиплексора обслуживает запросы.

Рисунок 2.1 – Схема алгоритма работы веб-сервера

3 ТЕХНОЛОГИЧЕСКАЯ ЧАСТЬ

В этом разделе будут выбраны средства реализации программного обеспечения. Будет выбран язык программирования. Будет приведена демонстрация работы программы.

3.1 Выбор средств реализации

В качестве языка программирования был выбран C в соответствии с заданием на курсовую работу. В качестве мультиплексора был выбран select в соответствии с заданием на курсовую работу. Для параллелизации обработки пользовательских запросов был использован пул потоков.

3.2 Реализация веб-сервера

Функция запуска сервера приведена в листинге 3.1.

Листинг 3.1 – Запуск сервера

```
1  int serve(uint16_t port)
2  {
3      tpool_t *tm = tpool_create(THREAD_POOL_CAPACITY);
4
5      int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
6      int reuse = 1;
7      if (setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR,
         &reuse, sizeof(reuse)) < 0) {
8          errorHandler(ERROR, "setsockopt() failed", "", 0);
9          return EXIT_FAILURE;
10     }
11
12     struct sockaddr_in serverAddress;
13     serverAddress.sin_family = AF_INET;
14     serverAddress.sin_port = htons(port);
15     serverAddress.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
16
17     int bound = bind(serverSocket, (struct sockaddr *)
         &serverAddress, sizeof(serverAddress));
18     if (bound != 0) {
```

```
19          char portString[5 + MAX_PORT_DIGITS];
20          sprintf(portString, "port %d", port);
21          errorHandler(ERROR, "Socket not bound", portString, 0);
22          return EXIT_FAILURE;
23      }
24
25      int listening = listen(serverSocket, BACKLOG);
26      if (listening < 0) {
27          printf("Error: The server is not listening.\n");
28          return EXIT_FAILURE;
29      }
30      report(&serverAddress);
31      int clientSocket;
32      fd_set client_fds;
33      while(1) {
34          FD_ZERO(&client_fds);
35          FD_SET(serverSocket, &client_fds);
36          if (select(serverSocket + 1,&client_fds, NULL,NULL,NULL)
              == -1)
37          {
38              errorHandler(ERROR, strerror(errno),  NULL, 0);
39              return EXIT_FAILURE;
40          }
41          if(FD_ISSET(serverSocket, &client_fds))
42          {
43              threadData_t *data = calloc(1, sizeof(threadData_t));
44              LogData *log = calloc(1, sizeof(*log));
45              log->clientAddr = calloc(1, sizeof(*log->clientAddr));
46              log->req = NULL;
47              clientSocket = acceptTCPConnection(serverSocket, log);
48              if (clientSocket >=0 )
49              {
50                  data->log = log;
51                  data->socket = clientSocket;
52                  tpool_add_work(tm, handleHTTPClient, data);
53              }
54          }
55      }
56      return 0;
57  }
```

Функция обработки соединения с клиентом представлена в листинге 3.2

Листинг 3.2 – Обработка клиентского соединения

```
1  void handleHTTPClient(void *arg)
2  {
3      threadData_t* data = arg;
4      int clientSocket = data->socket;
5      LogData* log = data->log;
6      char recvBuffer[INPUT_BUFFER_SIZE];
7      memset(recvBuffer, 0, sizeof(char) * INPUT_BUFFER_SIZE);
8
9      int nBytesReceived = recv(clientSocket, recvBuffer,
           sizeof(recvBuffer), 0);
10     if (nBytesReceived < 0) {
11         errorHandler(FORBIDDEN, "Failed to read request.", "",
               clientSocket);
12         freeLog(log);
13         free(data);
14         close(clientSocket);
15         return;
16     }
17     if (nBytesReceived < INPUT_BUFFER_SIZE) {
18         recvBuffer[nBytesReceived] = 0;
19     }
20
21     char *response = NULL;
22     char *filename = NULL;
23     char *req = NULL;
24     firstLine(recvBuffer, &req);
25     char *tmpReq = realloc(log->req, (strlen(req) *
           sizeof(*(log->req))) + 1);
26     if (tmpReq == NULL)
27     {
28         ErrorSystemMessage("Memory allocation: realloc()
               failure.");
29         free(response);
30         free(filename);
31         freeLog(log);
32         free(data);
33         close(clientSocket);
34         return;
35     }
36     log->req = tmpReq;
```

```
37      strcpy(log->req, req);
38      free(req);
39      if (router(recvBuffer, clientSocket, &filename) != 0)
40      {
41          free(response);
42          free(filename);
43          freeLog(log);
44          free(data);
45          close(clientSocket);
46          return;
47      }
48
49      ssize_t mimeTypeIndex = -1;
50      if ((mimeTypeIndex = fileTypeAllowed(filename)) == -1) {
51          free(filename);
52          errorHandler(FORBIDDEN, "mime type", "Requested file type
                not allowed.", clientSocket);
53          free(response);
54          freeLog(log);
55          free(data);
56          close(clientSocket);
57          return;
58      }
59
60      if (setResponse(filename, &response, OK, mimeTypeIndex,
            clientSocket, log) != 0) {
61          free(response);
62          freeLog(log);
63          free(data);
64          close(clientSocket);
65          return;
66      }
67
68      send(clientSocket, response, strlen(response) + 1, 0);
69      free(response);
70      free(filename);
71      logConnection(log);
72      free(data);
73      close(clientSocket);
74  }
```

3.3 Поддерживаемые запросы

Разработанный сервер может обработать GET и HEAD запросы. Когда клиент выполняет GET запрос, он получает в теле ответа запрошенный файл. Когда клиент выполняет HEAD запрос он в ответе получает только заголовки Content-Type и Content-Length. Ниже перечислены статусы ответов сервера, которые были реализованы.

— 200 — успешное завершение обработки запроса.

— 403 — доступ к запрошенному файлу запрещён, запрошен неподдерживаемый тип файла.

— 404 — запрашиваемый файл не найден.

— 405 — неподдерживаемый HTTP-метод (POST, PUT и т.д.).

В соответствии с заданием веб-сервер может отдавать файлы следующих форматов:

— html (text/html);

— css (text/css);

— js (text/javascript);

— png (image/png);

— jpg (image/jpg);

— jpeg (image/jpeg);

— gif (image/gif);

— svg (image/svg);

— swf (application/x-shockwave-flash).

4 ИССЛЕДОВАТЕЛЬСКАЯ ЧАСТЬ

В данном разделе будут представлены характеристики компьютера, на котором проводилось исследование. Также в данном разделе будут приведено описание и постановка исследования. Будут приведены результаты сравнения разработанного веб-сервера с NGINX [4].

4.1 Технические характеристики устройства

Технические характеристики устройства, на котором выполнялись измерения:

— процессор — AMD Ryzen 5 4600H с Radeon Graphics 3.00 ГГц [5];

— операционная система — Ubuntu 22.04 on WSL;

— версия ядра — 4.4.0-19041-Microsoft;

— оперативная память — 24 Гб;

— количество логических ядер — 12;

— ёмкость диска — 512 GB;

Во время тестирования ноутбук был включен в сеть питания и нагружен только приложениями встроенными в операционную систему и системой тестирования. Сторонние приложения запущены не были.

4.2 Описание исследования

Цель эксперимента — сравнить разработанный веб-сервер с NGINX по времени обработки различных запросов.

В ходе исследования анализируется время, затрачиваемое на отдачу файлов через разработанный веб-сервер и через NGINX.

Конфигурация NGINX представлена в листинге 4.1

Листинг 4.1 – Конфигурация nginx

```
events {
    worker_connections 1024;
}


http
{
    server {
        charset utf-8;
        listen 80;
        location / {
            root /mnt/c/zolot/CNCP/src;
            include /etc/nginx/mime.types;
            index index.html;
        }
    }
}
```

Для измерения времени обработки запросов использовалась утилита Apache Benchmark [6]

4.3 Результаты исследования

В таблице 4.1 приведены результаты нагрузочного тестирования. На каждом прогоне выполнялось 10000 запросов.

Таблица 4.1 – Среднее время (мс) обработки запроса через NGINX и через разработанный веб-сервер при обработке 10000 запросов

Размер файла	Число конкурентных запросов	веб-сервер	NGINX
612 байт	10	7.649	3.757
	100	53.438	34.555
	1000	354.575	214.196
441 Кбайт	10	10.911	6.701
	100	92.381	60.937
	1000	897.818	320.089

4.4 Вывод

Результаты тестирования показывают, что NGINX работает быстрее чем разработанный веб-сервер на 53%–180%. При обработке файла размером 612 байт при увеличении конкурентных запросов среднее время обработки запроса улучшилось на 50% по сравнению с NGINX. При обработке файла 441 Кбайт можно наблюдать абсолютно противоположную ситуацию. При увеличении числа конкурентных запросов среднее время обработки запроса веб-сервера по сравнению с NGINX ухудшилось на 127%.

Данные результаты можно объяснить тем, что разработанный сервер имеет более простую архитектуру и менее оптимизированную кодовую базу, чем NGINX. Также мультиплексор select, имеет ограничение на количество открытый файловых дескрипторов.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были решены все поставленные задачи и достигнута поставленная цель. Был разработан статический веб-сервер.

В ходе выполнения курсовой работы были проанализированы протокол HTTP, сокеты и мультиплексоры. Также было разработано программное обеспечение для обслуживания контента хранящегося во вторичной памяти.

Исследование показало, что NGINX в среднем обрабатывает запрос быстрее чем разработанный веб-сервер на 53–180%.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Веб-сервер: что это и для чего нужен [Электронный ресурс]. Режим доступа: https://gb.ru/blog/veb-server/?ysclid=lpz8olkf6v376427819 (дата обращения: 25.11.2023).

2. Обзор протокола HTTP [Электронный ресурс]. Режим доступа: https://developer.mozilla.org/ru/docs/Web/HTTP/Overview (дата обращения: 25.11.2023).

3. Стивенс Р., Раго С. Unix. Профессиональное программированиее, 2-е издание. – СПб.: Символ-Плюс, 2007. – 1040 с.

4. NGINX: Advanced Load Balancer, Web Server, Reverse Proxy [Электронный ресурс]. Режим доступа: https://www.nginx.com/ (дата обращения: 25.11.2023).

5. AMD Ryzen™ 5 4600H [Электронный ресурс]. Режим доступа: https://www.amd.com/ru/products/apu/amd-ryzen-5-4600h (дата обращения: 25.11.2023).

6. ab - Apache HTTP server benchmarking tool [Электронный ресурс]. Режим доступа: https://httpd.apache.org/docs/trunk/programs/ab.html (дата обращения: 25.11.2023).