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 ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Разработка статического сервера»

Студент **ИУ7-73Б** _____ **Светличная А. А.**

Руководитель курсовой работы _____ **Исполатов Ф. О.**

2023 г.

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)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ-7
_____ И. В. Рудаков
«16» сентября 2023 г.

З А Д А Н И Е
на выполнение курсовой работы

по теме

«Разработка статического сервера»

Студент группы **ИУ7-73Б**

Светличная Алина Алексеевна

Направленность КР

учебная

Источник тематики

НИР кафедры

График выполнения КР: 25% к 6 нед., 50% к 9 нед., 75% к 12 нед., 100% к 15 нед.

Техническое задание

Разработать статический веб-сервер для отдачи контента с диска. В качестве мультиплексора использовать pselect. Сервер должен реализовывать многопоточную обработку запросов с использованием пула потоков.

Оформление научно-исследовательской работы:

Расчетно-пояснительная записка на **12-20** листах формата А4.

Дата выдачи задания «16» сентября 2023 г.

Руководитель курсовой работы	_____	**Исполатов Ф. О.**
	(Подпись, дата)	(Фамилия И. О.)
Студент	_____	**Светличная А. А.**
	(Подпись, дата)	(Фамилия И. О.)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В свете постоянного развития веб-технологий и растущих требований к производительности веб-сервисов, актуальность создания собственного статического сервера набирает обороты в сфере веб-разработки. Статические веб-серверы представляют собой решение для обеспечения размещения и предоставления статических ресурсов, таких как файлы HTML, CSS, изображения, видео и другое, минимизируя при этом сложность серверных технологий.

Данная работа нацелена на подробное рассмотрение ключевых аспектов создания собственного статического сервера, начиная с фундаментальных принципов функционирования веб-серверов и завершая реализацией сервера на основе выбранных технологических подходов.

Цель: разработка статического сервера.

Для достижения поставленной цели необходимо решить ряд задач.

Задачи:

— провести анализ предметной области;

— спроектировать архитектуру прогормманого обеспечения;

— реализовать программное обеспчение с использованием требуемых технологий;

— провести исследование в области нагрузочного тестирования разработанного программного обеспечения.

1 Анализ предметной области

1.1 HTTP протокол

HTTP (HyperText Transfer Protocol) представляет собой протокол передачи информации в сети Интернет, целью которого является взаимодействие между клиентом и сервером. Изначально предназначенный для гипертекстовых документов, HTTP развился для передачи различных типов информации.

HTTP-запрос состоит из трех ключевых элементов:

— стартовая строка, определяющей параметры запроса или ответа;

— заголовки, предоставляющие информацию о передаче и другую вспомогательную информацию;

— тело (не всегда присутствует), содержащее передаваемые данные [1].

Стартовая строка дает серверу представление о запрошенном действии. Она структурирована следующим образом: *метод + URL + версия* [1].

Метод, или HTTP-глагол, определяет конкретное действие, которое требуется выполнить с страницей. Существует девять стандартных методов:

— GET — получение данных от сервера;

— HEAD — получение заголовков ответа без тела;

— POST — отправка данных на сервер для обработки или сохранения;

— PUT — обновление ресурса на сервере;

— PATCH — частичное обновление ресурса на сервере;

— DELETE — удаление указанного ресурса на сервере;

— CONNECT — установка туннеля к серверу через прокси;

— OPTIONS — запрос информации о возможностях сервера или характеристиках ресурса;

— TRACE — тестирование маршрута до указанного ресурса через прокси-серверы [1].

URL (Uniform Resource Locator) служит стандартизированным идентификатором ресурса, определяющим его точное местоположение. Именно с помощью URL записаны ссылки в интернете [1].

Версия указывает на версию протокола, которую необходимо использовать в ответе сервера [1].

HTTP-ответы следуют подобной структуре: *версия + код состояния + пояснение* [1].

Версия соответствует версии, указанной в запросе [1].

Код состояния указывает на статус запроса. Это трехзначное число, позволяющее определить, был ли запрос получен, обработан ли он, какие ошибки возникли [1].

В пояснении предоставляется краткое описание ответа [1].

Наиболее распространенные коды состояния и описания:

— 200 OK — запрос успешно обработан, сервер возвращает данные;

— 201 Created — ресурс успешно создан на сервере;

— 204 No Content — запрос успешен, но ответ не содержит тела;

— 400 Bad Request — сервер не может понять запрос из-за ошибок клиента;

— 401 Unauthorized — требуется аутентификация пользователя;

— 403 Forbidden — нет прав доступа к ресурсу;

— 404 Not Found — запрашиваемый ресурс не найден;

— 500 Internal Server Error — внутренняя ошибка сервера;

— 503 Service Unavailable — сервер временно не может обрабатывать запросы;

— 504 Gateway Timeout — время ожидания шлюза истекло [2].

Кроме того существует HTTPS протокол, являющийся расширением HTTP. Он вносит элемент безопасности для передаваемых данных. В то время как HTTP передает данные открыто, что облегчает их перехват, HTTPS обеспечивает безопасность данных с использованием SSL-сертификата. Этот сертификат сначала шифрует уязвимые данные на стороне клиента (например, в браузере), превращая их в случайную последовательность символов, и только затем отправляет на сервер [3].

1.2 Веб-сервер

Понятие «веб-сервер» может быть охарактеризовано как аппаратно, так и программно.

С аппаратной точки зрения веб-сервер представляет собой физическую машину, осуществляющую хранение файлов веб-сайта (таких как HTML-документы, CSS-стили, JavaScript-файлы, изображения и прочее) и их передачу конечному пользовательскому устройству (например, веб-браузеру). Этот компьютер подключен к сети Интернет и может быть доступен посредством доменного имени [4].

С программной точки зрения веб-сервер включает в себя несколько компонентов, регулирующих доступ веб-пользователей к размещенным на сервере файлам. Одним из базовых элементов является HTTP-сервер, ответственный за обработку URL-адресов (веб-адресов) и HTTP-протокола, используемого браузером для просмотра веб-страниц [4].

Упрощенно, при запросе браузером файла, размещенного на веб-сервере, браузер инициирует запрос через HTTP-протокол. При достижении запроса необходимого веб-сервера (компьютера), сервер HTTP (программное обеспечение) принимает запрос, находит запрошенный документ (или сообщает об ошибке) и передает его обратно, также посредством HTTP.

Для веб-сайта требуется статический или динамического веб-сервер.

Статический веб-сервер представляет собой компьютер с сервером HTTP. Термин «статика» применяется по причине отправки файлов браузеру в исходном виде [4].

Динамический веб-сервер включает в себя статический веб-сервер и дополнительное программное обеспечение, обычно сервер приложений и базу данных. Термин «динамика» используется, так как сервер приложений из-

меняет исходные файлы перед их отправкой в браузер пользователя через HTTP [4].

1.3 Сокеты

Сокет — универсальный интерфейс для создания каналов для межпроцессного взаимодействия [5].

Сокеты объединили в едином интерфейсе потоковую передачу данных подобную каналам pipe и FIFO и передачу сообщений, подобную очередям сообщений в System V IPC. Кроме того, сокеты добавили возможность создания клиент-серверного взаимодействия (один со многими) [5].

Интерфейс сокетов скрывает механизм передачи данных между процессами. В качестве нижележащего транспорта могут использоваться как внутренний транспорт в ядре Unix, так и практически любые сетевые протоколы. Для достижения такой гибкости используется перегруженная функция назначения сокету имени — bind(). Данная функция принимает в качестве параметров идентификатор пространства имён и указатель на структуру, которая содержит имя в соответствующем формате. Это могут быть имена в файловой системе Unix, IP адрес + порт в TCP/UDP, MAC-адрес сетевой карты в протоколе IPX [5].

Существует следующая классификация сокетов.

1. Stream — поток байтов без разделения на записи, подобный чтению-записи в файл или каналам в Unix. Процесс, читающий из сокета, не знает, какими порциями производилась запись в сокет пишущим процессом. Данные никогда не теряются и не перемешиваются [5].

   — Непрерывный поток байтов.

   — Упорядоченный приём данных.

   — Надёжная доставка данных.

2. Datagram — передача записей ограниченной длины. Записи на уровне интерфейса сокетов никак не связанны между собой. Отправка записей описывается фразой: "отправил и забыл". Принимающий процесс получает записи по отдельности в непредсказуемом порядке или не получает вовсе [5].

— Деление потока данных на отдельные записи.

— Неупорядоченный приём записей.

— Возможна потеря записей.

3. Sequential packets — надёжная упорядоченная передача с делением на записи. Использовался в Sequence Packet Protocol для Xerox Network Systems. Не реализован в TCP/IP, но может быть имитирован в TCP через Urgent Pointer [5].

— Деление потока данных на отдельные записи.

— Упорядоченная передача данных.

— Надёжная доставка данных.

4. Raw — предназначен для управление нижележащим сетевым драйвером. В Unix требует администраторских полномочий. Примером использования Raw-сокета является программа ping, которая отправляет и принимает управляющие пакеты управления сетью — ICMP [5].

Имена сокетов на сервере назначаются вызовом bind(), а на клиенте, как правило, генерируются ядром [5].

— Inet — сокеты именуются с помощью IP адресов и номеров портов.

— Unix — сокетам даются имена объектов типа socket в файловой системе.

— IPX — имена на основе MAC-адресов сетевых карт.

Для передачи данных с помощью семействе протоколов TCP/IP реализованы два вида сокетов Stream и Datagram. Все остальные манипуляции с сетью TCP/IP осуществляются через Raw-сокеты [5].

— TCP — Stream.

— UDP — Datagram.

— ICMP — RAW.

— Sequential packets — были экспериментальные реализации в 1990-х, которые не вышли за рамки научных исследований.

1.4 Мультиплексирование ввода/вывода

Мультиплексирование ввода/вывода (I/O multiplexing) — метод, позволяющий одному процессу эффективно управлять несколькими операциями ввода/вывода без необходимости создания множества потоков или процессов.

Основной инструмент для мультиплексирования ввода/вывода — это механизмы select/pselect/poll/epoll.

select — это один из старейших системных вызовов для мультиплексирования ввода/вывода, был введен в ранних версиях UNIX.

Принимает три списка файловых дескрипторов: для чтения (readfds), записи (writefds), и исключений (exceptfds). Блокирует выполнение программы до тех пор, пока не произойдет событие на хотя бы одном из указанных файловых дескрипторов [6].

Существует ограничение в количестве отслеживаемых дескрипторов (обычно 1024) и неэффективен при большом числе отслеживаемых дескрипторов из-за линейного поиска [6].

pselect — предоставляет дополнительные возможности по сравнению с select и является улучшением этого последнего.

pselect может блокировать сигналы, которые указаны в наборе сигналов (sigmask). Это позволяет избежать прерываний в момент ожидания событий [7].

Как и select, pselect ожидает определенного времени до возврата из блокировки. Однако, вместо структуры timeval, pselect использует структуру timespec, что позволяет указывать более длительные интервалы времени и обеспечивает более высокую точность [7].

poll — более современный вариант select, который не имеет ограничений на количество отслеживаемых дескрипторов.

Принимает массив структур pollfd, каждая из которых содержит информацию о файловом дескрипторе и ожидаемых событиях. Блокирует выполнение программы до наступления события на хотя бы одном из отслеживаемых дескрипторов [6].

epoll — это механизм, специфичный для ядра Linux, который предоставляет высокопроизводительный способ отслеживания событий ввода/вывода. Имеет три системных вызова: epoll_create, epoll_ctl, и epoll_wait. Использует объекты epoll_event для представления событий на файлах. Позво-

ляет масштабировать на большое количество отслеживаемых дескрипторов [6].

1.5 Параллельная обработка запросов

Thread pool и prefork — стратегии, которые используются в многозадачных средах для обработки параллельных задач. Они оба нацелены на улучшение производительности, но используют различные подходы.

Thread pool — пул потоков, представляющий собой набор заранее созданных потоков, которые могут выполнять различные задачи. Основная идея состоит в том, чтобы избежать создания и уничтожения потоков каждый раз при поступлении задачи. Вместо этого потоки находятся в пуле и могут многократно использоваться для выполнения различных задач [8].

Преимущества Thread pool:

— эффективное управление ресурсами — не происходит многократного создания и уничтожения потоков;

— повышение отзывчивости — поскольку потоки уже созданы, они могут немедленно начать выполнение задачи [8].

Prefork — это стратегия, основанная на создании нескольких процессов перед тем, как они начнут обрабатывать запросы. Этот подход часто используется в веб-серверах, таких как Apache с модулем MPM (Multi-Processing Module).

Преимущества Prefork:

— изоляция задач — каждый процесс работает в своем собственном адресном пространстве, что предотвращает взаимное влияние задач;

— надежность — если один из процессов выходит из строя, другие процессы остаются несвязанными с ним.

Вывод

В данной главе была проанализированы и описаны основные механизмы необхдимые для дальнейшей разработки статического сервера, а именно протокол HTTP, механизмы мультиплесирования ввода/вывода параллелизации обработки запросов.

2 Конструкторская часть

Схемы алгоритмов

На рисунке 2.1 изображена концептуальная схема работы будущего программного обеспечения.

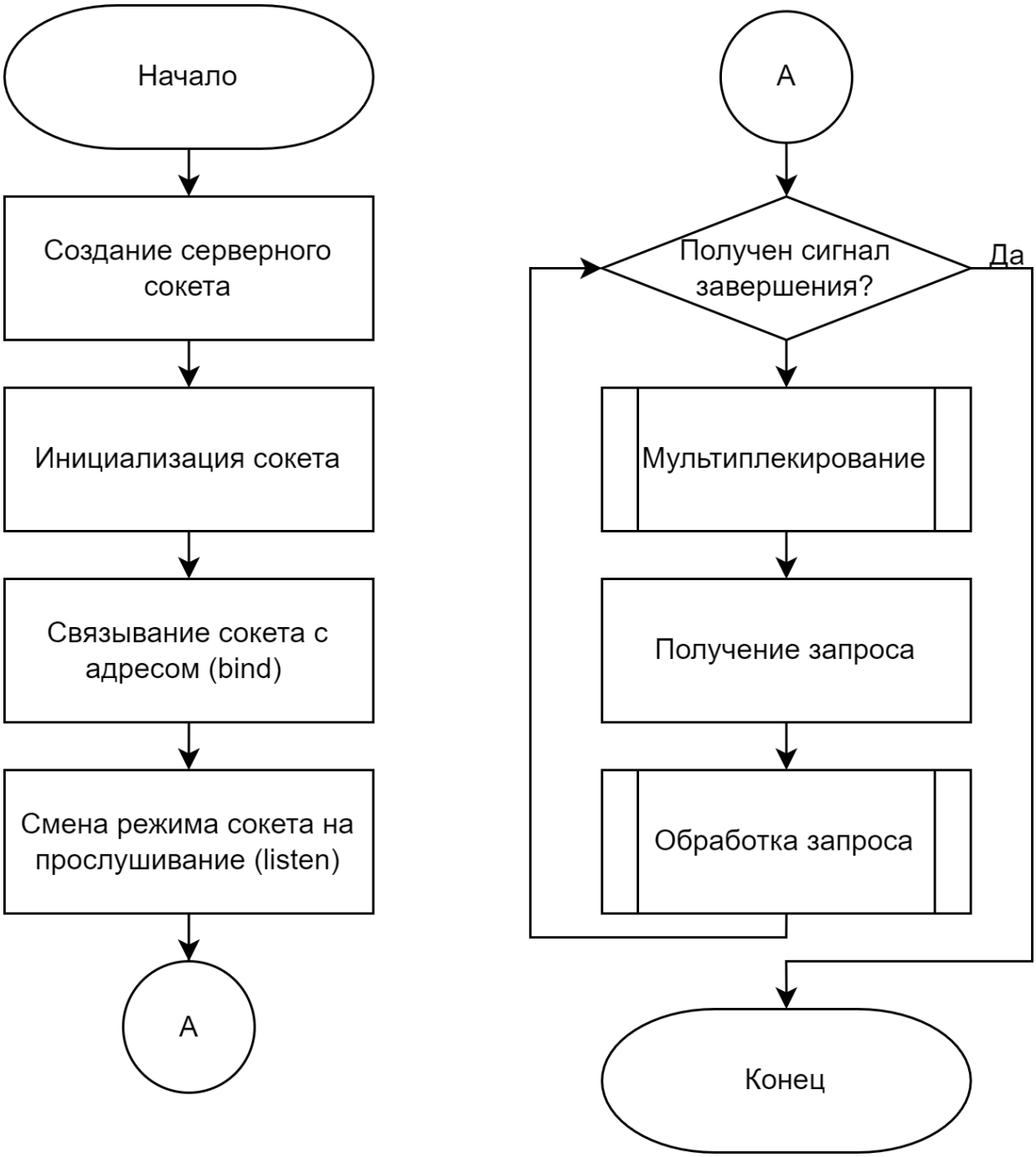

Рисунок 2.1 – Обобщенная схема программного обеспечения

Вывод

В данной главе была представлена общая схема разрабатываемого программного обеспечения.

3 Технологическая часть

3.1 Средства реализации

Для реализации программного обеспечения в сответствии требованиям к курсовому проекту и варианту был выбран язык C (без использования сторонних библиотек), механизм мультиплексирования клиентских соединений — pselect, механизм параллельной обработки запросов — thread pool.

3.2 Листинги реализации

Листинг 3.1 – Создание и запуск сервера

```
1  int main() {
2
3      setlocale(LC_ALL, "Russian");
4      WSADATA wsa;
5      SOCKET server_socket, client_socket;
6      struct sockaddr_in server_addr, client_addr;
7      int client_addr_len = sizeof(client_addr);
8
9      if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) {
10         perror("Error: initialization WinSock");
11         return 1;
12     }
13
14     server_socket = socket(AF_INET, SOCK_STREAM, 0);
15     if (server_socket == INVALID_SOCKET) {
16         perror("Error: creating socket");
17         return 1;
18     }
19
20     server_addr.sin_family = AF_INET;
21     server_addr.sin_addr.s_addr = INADDR_ANY;
22     server_addr.sin_port = htons(PORT);
23
```

```
24      if (bind(server_socket, (struct sockaddr *)&server_addr,
            ↪ sizeof(server_addr)) == SOCKET_ERROR) {
25          perror("Error: binding socket");
26          return 1;
27      }
28
29      if (listen(server_socket, 10) == SOCKET_ERROR) {
30          perror("Error: listening socket");
31          return 1;
32      }
33
34      ...
35
36      closesocket(server_socket);
37      WSACleanup();
38
39      return 0;
40  }
```

Листинг 3.2 – Обработка запроса

```
1   #define MAX_FILE_SIZE 100 * 1024 * 1024
2
3   char ROOT[] = "./www";
4
5   const char *supported_file_types[] = {
6       ".html", ".css", ".js", ".png", ".jpg", ".jpeg", ".swf",
7       ".gif", NULL
8   };
9
10  int is_supported_file_type(const char *path) {
11      const char *extension = strrchr(path, '.');
12      if (extension != NULL) {
13          for (int i = 0; supported_file_types[i]; i++) {
14              if (strcasecmp(extension, supported_file_types[i])
```

```
            ↪ == 0) {
                return 1;
            }
        }
    }
    return 0;
}

const char* get_content_type(const char *path) {
    const char *extension = strrchr(path, '.');
    if (extension != NULL) {
        if (strcasecmp(extension, ".html") == 0)
            return "text/html";
        if (strcasecmp(extension, ".css") == 0)
            return "text/css";
        if (strcasecmp(extension, ".js") == 0)
            return "application/javascript";
        if (strcasecmp(extension, ".png") == 0)
            return "image/png";
        if (strcasecmp(extension, ".jpg") == 0 || strcasecmp(
          ↪ extension, ".jpeg") == 0)
            return "image/jpeg";
        if (strcasecmp(extension, ".swf") == 0)
            return "application/x-shockwave-flash";
        if (strcasecmp(extension, ".gif") == 0)
            return "image/gif";
    }
    return "application/octet-stream";
}

void handle_request(SOCKET client_socket) {
    char request[1024];
    int bytes_received = recv(client_socket, request, sizeof(
      ↪ request), 0);
```

```
46      if (bytes_received <= 0) {
47          printf("Error: receiving data\n");
48          return;
49      }
50      request[bytes_received] = '\0';
51      printf("\nRequest received: %s", request);
52
53      char method[255];
54      char path[1024];
55
56      sscanf(request, "%s %s", method, path);
57
58      if (strcmp(path, "/") == 0) {
59          strncpy(path, "/main.html", sizeof(path) - 1);
60      }
61
62      char full_path[1024];
63          snprintf(full_path, sizeof(full_path), "%s%s", ROOT,
              ↪ path);
64
65      struct stat file_stat;
66      if (stat(full_path, &file_stat) == -1 || file_stat.st_size
          ↪ > MAX_FILE_SIZE) {
67          send_response(client_socket,
68              "HTTP/1.1 404 Not Found\r\n\r\n");
69      } else if (!is_supported_file_type(full_path)) {
70          send_response(client_socket,
71              "HTTP/1.1 403 Forbidden\r\n\r\n");
72      } else {
73          int file_fd = open(full_path, O_RDONLY);
74          if (file_fd == -1) {
75              send_response(client_socket,
76                  "HTTP/1.1 404 Not Found\r\n\r\n");
77          }
```

```
78                  else if (strcmp(method, "GET") == 0 || strcmp(
                        ↪ method, "HEAD") == 0)
79                  {
80                          const char *content_type =
                            ↪ get_content_type(full_path);
81                          char response_header[512];
82                          snprintf(response_header, sizeof(
                            ↪ response_header), "HTTP/1.1 200 OK\r
                            ↪ \nContent-Length: %ld\r\nContent-
                            ↪ Type: %s\r\n\r\n", file_stat.st_size
                            ↪ , content_type);
83                          send_response(client_socket,
                            ↪ response_header);
84
85                          if (strcmp(method, "GET") == 0)
86                          {
87                              char buffer[1024];
88                              ssize_t bytes_read;
89                              ssize_t total_bytes_read = 0;
90
91                              int file_fd = open(full_path, O_BINARY
                                ↪  | O_RDONLY);
92                              while ((bytes_read = read(file_fd,
                                ↪ buffer, sizeof(buffer))) > 0) {
93                                  ssize_t total_sent = 0;
94                                  total_bytes_read += bytes_read;
95                                  while (total_sent < bytes_read) {
96                                      ssize_t sent = send(
                                        ↪ client_socket, buffer +
                                        ↪ total_sent, bytes_read -
                                        ↪ total_sent, 0);
97                                      if (sent == -1) {
98                                          perror("Error: sending data
                                            ↪ ");
```

```
                                    close(file_fd);
                                    return;
                                }
                                total_sent += sent;
                            }
                        }
                        close(file_fd);
                        printf("Total read from file: %ld\n",
                            ↪ total_bytes_read);
                    }

            close(file_fd);
        }
                else
                {
                send_response(client_socket, "HTTP/1.1 405 Method
                    ↪  Not Allowed\r\n\r\n");
                return;
        }
    }

    shutdown(client_socket, SD_SEND);
}
```

Листинг 3.3 – Пул потоков

```
typedef struct SocketQueue {
    SOCKET sockets[1024];
    int head;
    int tail;
    pthread_mutex_t mutex;
    pthread_cond_t condition;
} SocketQueue;

SocketQueue socketQueue;
```

```
10
11 void SocketQueue_init(SocketQueue *queue) {
12     queue->head = 0;
13     queue->tail = 0;
14     pthread_mutex_init(&(queue->mutex), NULL);
15     pthread_cond_init(&(queue->condition), NULL);
16 }
17
18 void SocketQueue_push(SocketQueue *queue, SOCKET s) {
19     pthread_mutex_lock(&(queue->mutex));
20     queue->sockets[queue->tail++] = s;
21     pthread_cond_signal(&(queue->condition));
22     pthread_mutex_unlock(&(queue->mutex));
23 }
24
25 SOCKET SocketQueue_pop(SocketQueue *queue) {
26     pthread_mutex_lock(&(queue->mutex));
27     while (queue->head == queue->tail) {
28         pthread_cond_wait(&(queue->condition), &(queue->mutex));
29     }
30     SOCKET s = queue->sockets[queue->head++];
31     pthread_mutex_unlock(&(queue->mutex));
32     return s;
33 }
34
35 void *worker_thread(void *arg) {
36     while (1) {
37         SOCKET client_socket = SocketQueue_pop(&socketQueue);
38         handle_request(client_socket);
39         closesocket(client_socket);
40     }
41     return NULL;
42 }
43
```

```
44 void *worker_thread(void *arg) {
45     while (1) {
46         SOCKET client_socket = SocketQueue_pop(&socketQueue);
47         handle_request(client_socket);
48         closesocket(client_socket);
49     }
50     return NULL;
51 }
52
53 int main() {
54
55     ...
56
57     SocketQueue_init(&socketQueue);
58     pthread_t threadPool[THREAD_POOL_SIZE];
59     for (int i = 0; i < THREAD_POOL_SIZE; i++) {
60         pthread_create(&(threadPool[i]), NULL, worker_thread,
           ↪ NULL);
61     }
62
63     fd_set readfds;
64     while (1) {
65         FD_ZERO(&readfds);
66         FD_SET(server_socket, &readfds);
67
68         int activity = select(server_socket + 1, &readfds, NULL,
           ↪  NULL, NULL);
69
70         if (activity < 0) {
71             perror("Error: checking file descriptors");
72             continue;
73         }
74
75         if (FD_ISSET(server_socket, &readfds)) {
```

```
76              client_socket = accept(server_socket, (struct
                    ↪ sockaddr *)&client_addr, &client_addr_len);
77              if (client_socket == INVALID_SOCKET) {
78                  perror("Error: accepting connection");
79                  continue;
80              }
81              SocketQueue_push(&socketQueue, client_socket);
82          }
83      }
84
85      for (int i = 0; i < THREAD_POOL_SIZE; i++) {
86          pthread_join(threadPool[i], NULL);
87      }
88
89      ...
90
91      return 0;
92  }
```

3.3 Функционал разработанного программного обеспечения

Функционал разработанного программного обеспечения соответствкет требованиям к курсовому проекту:

— поддерживаются запросы GET и HEAD;

— поддерживаются коды состояний 200, 403, 404, 405;

— доступны форматы фойлов html, css, js, png, jpg, jpeg, mp4, gif (с выставлением соответсвтующего conten-type);

— доступна передача фойлов до 100 мб;

— по умолчанию возвращается базовая html-страница;

— невозможен выход за корневой католог;

— доступен логгер.

Вывод

В данной главе была представлена листинги реализации основных частей программного обеспечения, а также описан доступный функционал, который удовлетворяет требованиям задания на курсовой проект.

4 Исследовательская часть

Сравнение нагрузочного тестирования

По требованием курсовой работы необходимо провести сравнение нагрузочного тестирование разработанного статического сервера и NGINX-сервера, используя механизм Apache Benchmark. Конфигурация NGINX представлена далее.

Листинг 4.1 – Конфигурация Nginx

```
worker_processes auto;

error_log logs/error.log;

events {
    worker_connections 1024;
}

http {
        server {
                listen 80;
                server_name localhost;

                location / {
                }
        }
}
```

Критерием сравнения будет выстпуать количество запросов в секунду.

Таблица 4.1 – Результаты измерений

Кол-во запросов	Кол-во клиентов	Размер	Сервер	NGINX
100	1	1 КБ	46.32	46.16
100	10	1 КБ	47.69	46.99
1000	1	1 КБ	305.45	296.90
1000	10	1 КБ	379.84	335.69
100	1	1 МБ	24.56	43.38
100	10	1 МБ	32.80	44.45
1000	1	1 МБ	45.74	233.12
1000	10	1 МБ	113.71	244.78
100	1	100 МБ	0.51	7.37

Вывод

В данной главе была проведено исследование на тему сравнения нагрузочного тестирования собственного статического сервера и сервера NGINX. Результаты показали, что при малых размерах передаваемях файлов разница между данными серверами не более 10%, однако с возрастанием размеров файлов разрыв результатов растет, являясь 50% для файла 1МБ и 90% для файла 100 МБ.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы были реализован собственный статический сервер и сравнена его роботоспособность с популярным решением NGINX. Результаты показали, что NGINX обрабатывает большее количество запросов в секунду, а также данная пропорциональна размерам передаваемых файлов

Цель, поставленная в начале работы, была достигнута. В ходе ее выполнения были решены все задачи:

— проведен анализ предметной области;

— спроектирована архитектура программного обеспечения;

— реализовано программное обеспечение с использованием требуемых технологий;

— проведено исследование в области нагрузочного тестирования разработанного программного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Обзор протокола HTTP [Электронный ресурс]. URL: https://developer.mozilla.org/ru/docs/Web/HTTP/Overview (дата обращения: 12.12.2023).

2. Коды состояния ответа HTTP [Электронный ресурс]. URL: https://developer.mozilla.org/ru/docs/Web/HTTP/Status (дата обращения: 12.12.2023).

3. В чем разница между HTTP и HTTPS? [Электронный ресурс]. URL: https://aws.amazon.com/ru/compare/the-difference-between-https-and-http/ (дата обращения: 12.12.2023).

4. Что такое веб-сервер [Электронный ресурс]. URL: https://developer.mozilla.org/ru/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server (дата обращения: 12.12.2023).

5. Сокеты [Электронный ресурс]. URL: https://parallel.uran.ru/book/export/html/498 (дата обращения: 12.12.2023).

6. select vs. poll vs. epoll [Электронный ресурс]. URL: https://www.hackingnote.com/en/versus/select-vs-poll-vs-epoll/ (дата обращения: 12.12.2023).

7. pselect [Электронный ресурс]. URL: https://www.hackingnote.com/en/versus/select-vs-poll-vs-epoll/https://www.opennet.ru/man.shtml?topic=pselect&category=2&russian=0 (дата обращения: 12.12.2023).

8. Thread Pools [Электронный ресурс]. URL: https://www.opennet.ru/docs/RUS/glib_api/glib-Thread-Pools.html (дата обращения: 12.12.2023).