



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, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

---

# РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

## *К КУРСОВОЙ РАБОТЕ*

*НА ТЕМУ:*

*«Статический веб-сервер на языке С»*

Студент ИУ7-71Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

Корниенко К. Ю.  
(И. О. Фамилия)

Руководитель курсовой работы

\_\_\_\_\_  
(Подпись, дата)

Яковидис Н. О.  
(И. О. Фамилия)

*2023 г.*

# СОДЕРЖАНИЕ

|  |           |
|--|-----------|
| <b>ВВЕДЕНИЕ</b>  | <b>4</b>  |
| <b>1 Аналитический раздел</b>                                    | <b>5</b>  |
| 1.1 Сокеты BSD . . . . .   | 5         |
| 1.2 Стек протоколов TCP/IP . . . . .                             | 7         |
| 1.3 Протокол HTTP . . . . .                                      | 8         |
| 1.3.1 Методы HTTP . . . . .                                      | 9         |
| <b>2 Конструкторский раздел</b>                                  | <b>10</b> |
| 2.1 Пул потоков . . . . .  | 10        |
| <b>3 Технологический раздел</b>                                  | <b>14</b> |
| 3.1 Выбор средств разработки . . . . .                           | 14        |
| 3.2 Сборка проекта . . . . .                                     | 14        |
| <b>4 Исследовательский раздел</b>                                | <b>24</b> |
| 4.1 Анализ производительности программного обеспечения . . . . . | 24        |
| <b>ЗАКЛЮЧЕНИЕ</b>  | <b>26</b> |
| <b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>                          | <b>27</b> |
| <b>ПРИЛОЖЕНИЕ А</b>  | <b>28</b> |

# ВВЕДЕНИЕ

С развитием информационных технологий компьютерные сети играют ключевую роль в современном мире. Они обеспечивают связь между компьютерами, серверами, мобильными устройствами и другими сетевыми устройствами, обеспечивая передачу данных и обмен информацией. В связи с этим актуальной является проблема обеспечения безопасности и эффективности работы компьютерных сетей.

Целью курсовой работы является изучение основных принципов разработки веб-серверов с использованием механизмов, предоставленных ядром операционной системы, взаимодействие с которым осуществляется на языке С.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- провести анализ предметной области;
- спроектировать и реализовать сервер;
- провести сравнение разработанного сервера с nginx.

# 1 Аналитический раздел

Веб-сервер — это программное средство, обеспечивающее пользователям сети доступ к гипертекстовым документам, расположенным на конкретном веб-узле [1]. Веб-сервер принимает HTTP [2] запросы от клиентов и отправляет им HTTP-ответы. Наиболее популярными веб-серверами являются Apache и Nginx [3].

Когда запрос от клиента поступает на веб-сервер, сервер использует специальные модули и конфигурации для обработки запроса и передачи статических файлов, таких как HTML, CSS, JavaScript, изображения и другие ресурсы.

Обычно веб-сервер на Linux предоставляет статические файлы, используя свои встроенные функции и механизмы обработки запросов, а также взаимодействует с ядром операционной системы для чтения файлов с диска и передачи их по сети клиенту. С помощью специальных модулей и конфигураций веб-сервер настраивается для эффективной отдачи статики [3].

## 1.1 Сокеты BSD

Для передачи данных между множеством потоков, запущенных на одном или нескольких компьютерах, связанных между собой сетью, используется стек протоколов TCP/IP [4]. Сокеты представляют собой абстракцию конечной точки сетевого взаимодействия, которая, в случае TCP/IP, объединяет в виде структуры IP-адрес и номер порта [5].

Сокеты были впервые введены в UNIX BSD как универсальное средство взаимодействия процессов как на отдельно взятой машине, так и в распределенных системах. Сокет создается системным вызовом `socket()`. На рисунке 1.1 представлена упрощенная схема взаимодействия двух процессов с помощью сокетов.

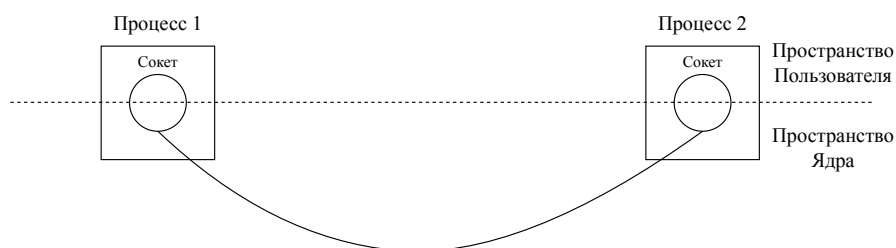


Рисунок 1.1 – Схема взаимодействия процессов с помощью сокетов

На рисунке 1.2 представлена последовательность системных вызовов, используемых на стороне клиента и на стороне сервера для обмена данными между процессами.

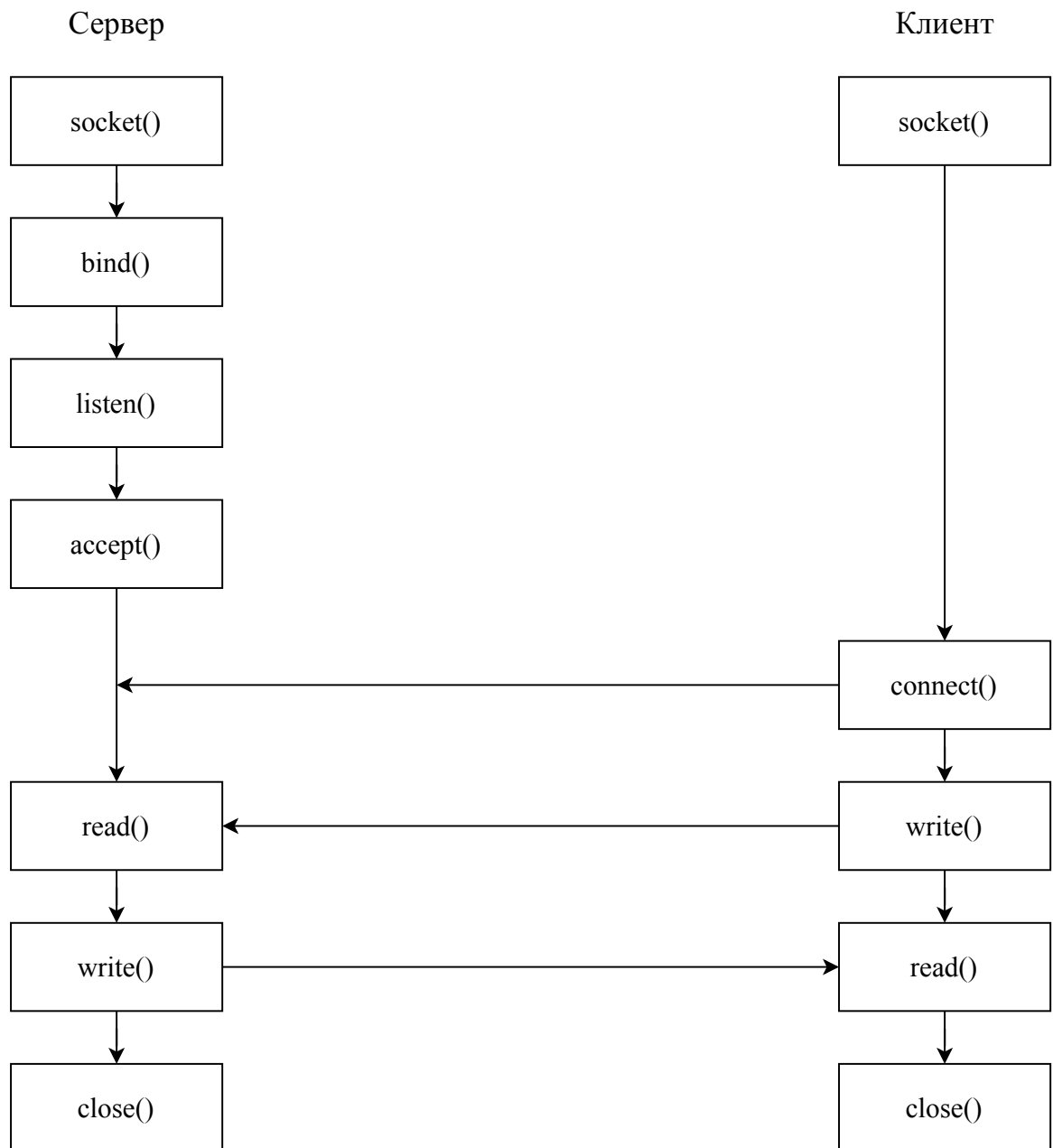


Рисунок 1.2 – Системные вызовы для взаимодействия процессов с помощью сокетов

Адреса и номера портов сокетов BSD должны быть указаны в сетевом порядке байтов [5].

## 1.2 Стек протоколов TCP/IP

Обычно общий термин «TCP/IP» означает все, что связано с конкретными протоколами TCP и IP. Это может включать другие протоколы, приложения, а также среду сети. Примерами таких протоколов могут быть UDP, ARP и ICMP. Примерами таких приложений могут быть TELNET, FTP и rcp.

Протокол TCP предназначен для разбивки сообщения на дейтаграммы и соединения их в конечной точке маршрута. TCP обеспечивает надежную, упорядоченную и проверенную доставку потока восьмибитовых байтов между приложениями, работающими на хостах, сообщающихся с помощью IP-сети [6].

Протокол IP (Internet Protocol - межсетевой протокол) является маршрутизируемым протоколом сетевого уровня стека протоколов TCP/IP. Этот протокол предназначен для объединения отдельных компьютерных сетей во всемирную сеть Интернет. Самой важной функцией этого протокола является адресация сети [6]. Задачей протокола IP является доставка пакетов от исходного узла на предназначенный узел только с помощью функции IP-адресация в заголовках пакетов. Для этого, IP определяет структуры пакетов, которые инкапсулируют данные. Он также определяет методы адресации, которые используются, чтобы обозначать дейтаграмму с исходными и предназначенными информацией. IP отвечает за адресации хостов и для маршрутизации пакета от исходного узла к предназначенному узлу через один или несколько IP-сетей.

На рисунке 1.3 приведены состояния сеанса TCP.

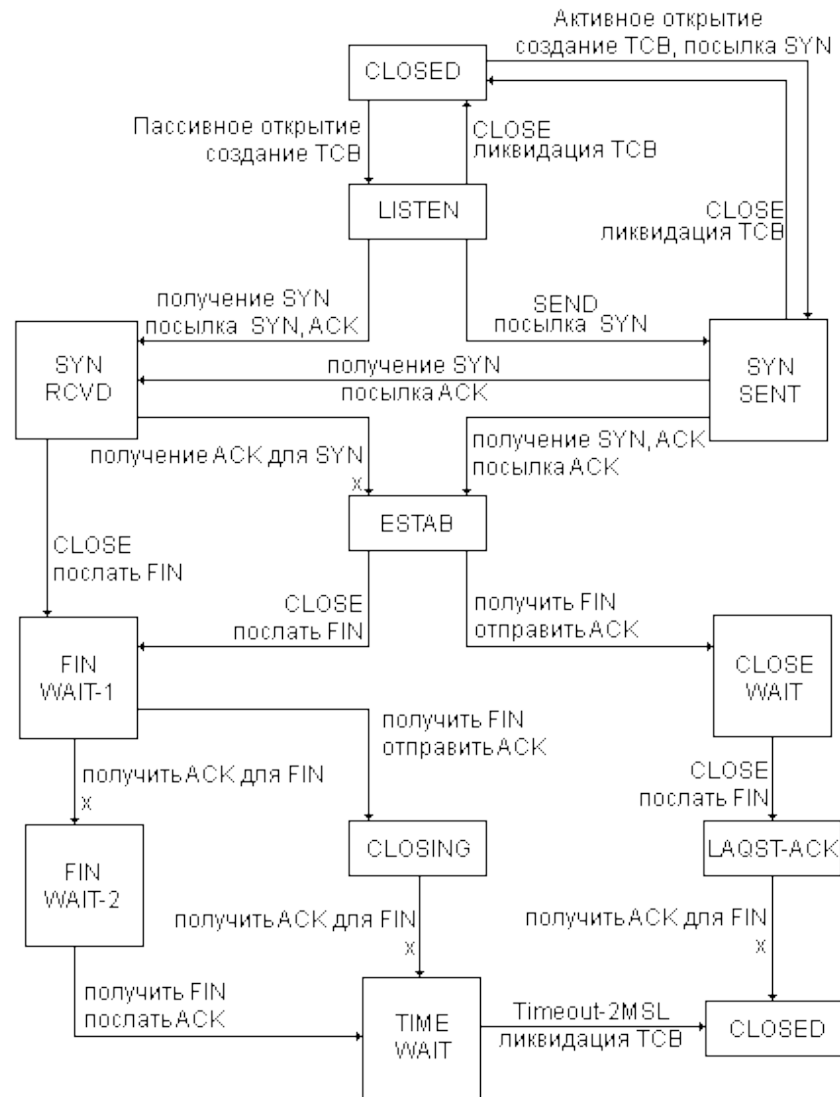


Рисунок 1.3 – Состояния сеанса TCP

## 1.3 Протокол HTTP

HTTP (англ. HyperText Transfer Protocol) — протокол прикладного уровня модели OSI/ISO, предназначенный для передачи гипертекстовых документов [7]. Протокол HTTP предполагает использование клиент-серверной структуры передачи данных. Клиентское приложение формирует запрос и отправляет его на сервер, после чего серверное программное обеспечение обрабатывает данный запрос, формирует ответ и передаёт его обратно клиенту. После этого клиентское приложение может продолжить отправлять другие запросы, которые будут обработаны аналогичным образом.

Так как HTTP — клиент-серверный протокол, то запросы отправляются какой-то одной стороной (участником обмена, либо прокси). Чаще всего в качестве участника обмена выступает веб-браузер.

Каждый запрос отправляется серверу, обрабатывающему его и возвращающему ответ. Между клиентом и сервером на пути послыки запросов и ответов может быть множество посредников: определенных сетевых устройств, например, маршрутизаторов. Все эти посредники оперируют на сетевом и транспортном уровнях.

### 1.3.1 Методы HTTP

Метод определяет операцию, которую нужно осуществить с указанным ресурсом. Спецификация HTTP 1.1 не ограничивает количество методов, однако используются лишь некоторые, наиболее стандартные методы:

- OPTIONS — запрос методов сервера (Allow);
- GET — запрос документа;
- HEAD — аналог GET, но без тела ответа;
- POST — передача данных от клиента;
- PUT — размещение файла по URI/изменение данных;
- PATCH — частичное изменение информации на веб-сервере;
- DELETE — удаление файла по URI/удаление данных.

URI (англ. Uniform Resource Identifier) — путь до конкретного ресурса, над которым необходимо осуществить операцию (метод HTTP).



## 2 Конструкторский раздел

### 2.1 Пул потоков

На рисунке 2.1 представлена концептуальная схема работы пула потоков.

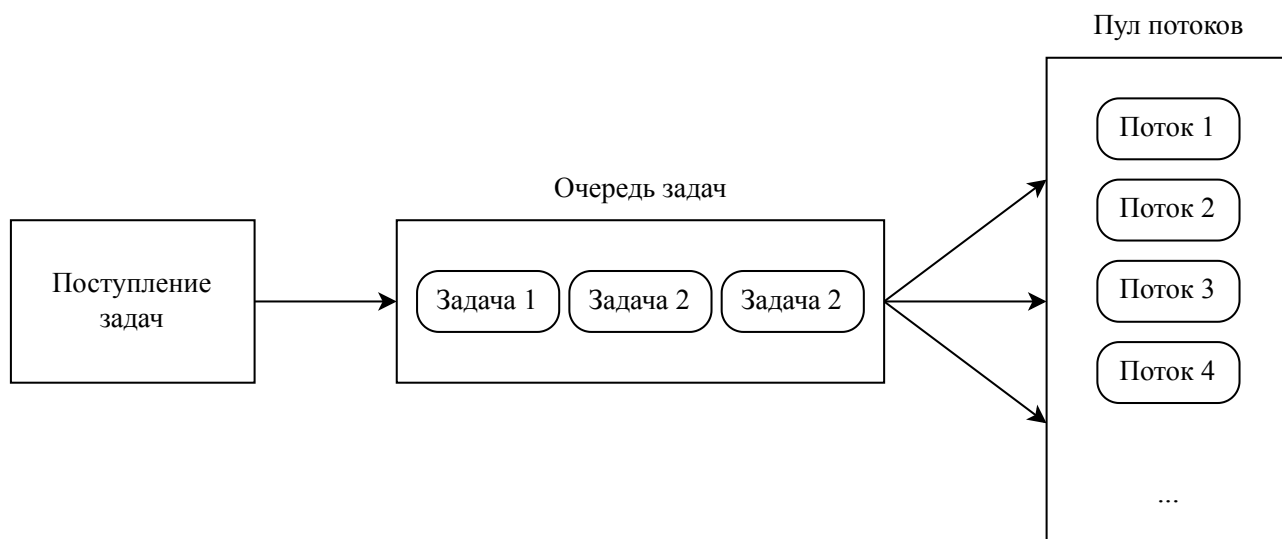


Рисунок 2.1 – Концептуальная схема работы пула потоков

На рисунке 2.2 представлена структура HTTP-сообщения.

| Метод                                | Путь | Протокол |
|--------------------------------------|------|----------|
| Заголовки                            |      |          |
| (Пустая строка)                      |      |          |
| Тело сообщения (может отсутствовать) |      |          |

Рисунок 2.2 – Структура HTTP-сообщения

На рисунке 2.3 представлена схема работы главного модуля сервера.



Рисунок 2.3 – Схема работы главного модуля сервера

На рисунке 2.4 представлена схема работы парсера запросов.

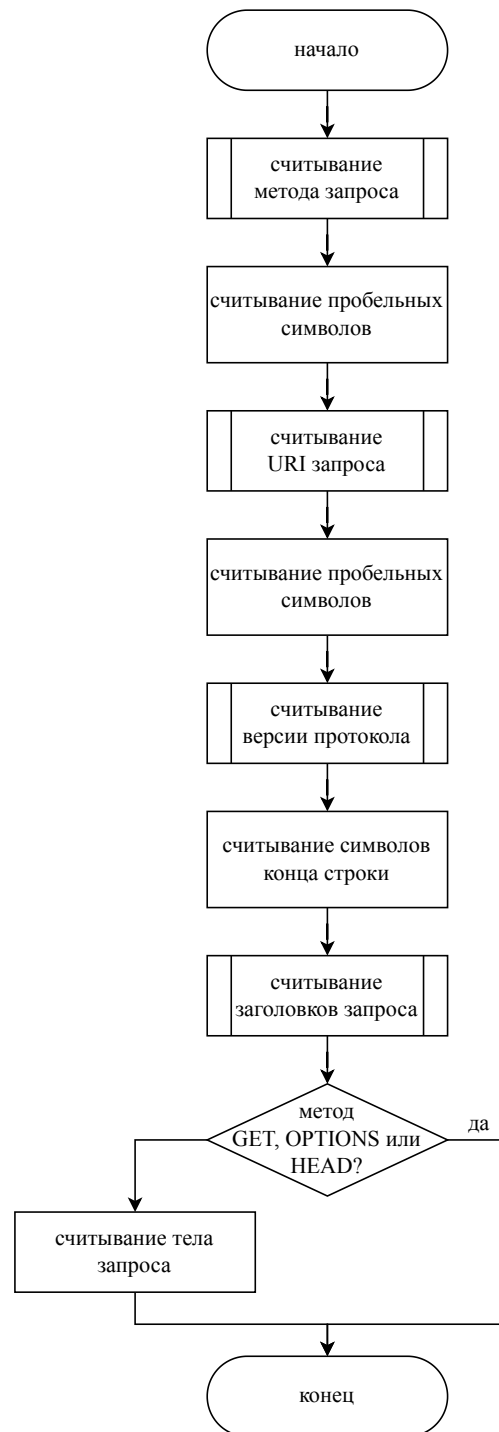


Рисунок 2.4 – Схема работы парсера запросов

На рисунке 2.5 представлена схема обработки GET-запроса.

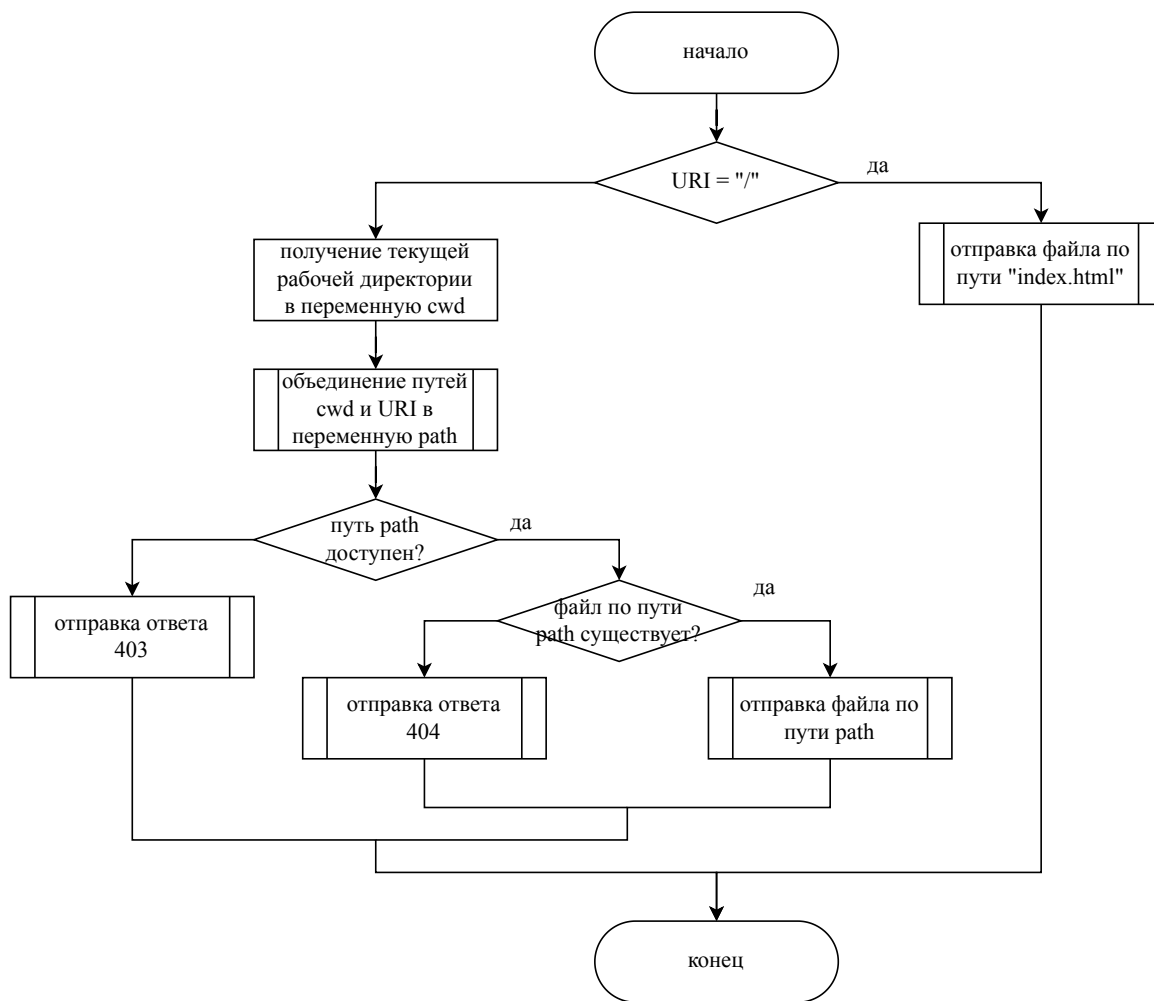


Рисунок 2.5 – Схема обработки GET-запросов

## 3 Технологический раздел

### 3.1 Выбор средств разработки

Для реализации статического веб-сервера был выбран язык С в соответствии с требованиями курсовой работы. Для выполнения задач курсового проекта были использованы системные вызовы, предоставляемые ядром операционной системы Linux.

Для сборки проекта был написан Makefile-скрипт.

### 3.2 Сборка проекта

На листинге 3.1 представлен Makefile-скрипт для сборки проекта.

Листинг 3.1 – Скрипт для сборки проекта

```
CC = gcc
CFLAGS = -Iinc -g -ggdb
SRC = $(wildcard src/*.c)
OBJ = $(SRC:src/%.c=build/%.o)

build: $(OBJ)
    $(CC) -o build/app $^

build/%.o: src/%.c
    mkdir -p $(@D)
    $(CC) $(CFLAGS) -c $< -o $@

run:
    cd static && ../build/app

clean:
    rm -rf build

.PHONY: build clean
```

На листинге 3.2 представлен код основного модуля сервера.

Листинг 3.2 – Код основного модуля сервера

```
#include "server.h"
#include "logger.h"

int main(void)
{
    set_logger_level(LOG_LEVEL_INFO);
    set_logger_file("../server.log");
    set_logger_redirect(LOG_REDIRECT_STDERR | LOG_REDIRECT_FILE);

    init_server("127.0.0.1", 8080);

    return run_server();
}
```

На листинге 3.3 представлен код работы с сервером.

Листинг 3.3 – Код работы с сервером

```
#include <arpa/inet.h>
#include <errno.h>
#include <string.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#include "logger.h"
#include "server.h"
#include "request_queue.h"
#include "thread_pool.h"

#define QUEUE_CAPACITY 10
#define NUM_WORKER_THREADS 4

static struct
{

```

```

    struct sockaddr_in addr;
    int queue_capacity;
    int num_worker_threads;
    bool running;
} server;

static void interrupt_handler(int sig)
{
    server.running = false;
    LOG_INFO("server stopped");
}

void init_server(const char* ip_address, int port)
{
    server.addr.sin_family = AF_INET;
    server.addr.sin_addr.s_addr = inet_addr(ip_address);
    server.addr.sin_port = htons(port);
    server.queue_capacity = QUEUE_CAPACITY;
    server.num_worker_threads = NUM_WORKER_THREADS;
    server.running = false;
}

int run_server(void)
{
    // create socket and bind it to address
    int server_socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket_fd == -1) {
        LOG_ERROR("socket: %s", strerror(errno));
        return EXIT_FAILURE;
    }

    if (setsockopt(server_socket_fd, SOL_SOCKET, SO_REUSEADDR,
        &(int){1},
        sizeof(int)) == -1) {
        LOG_ERROR("setsockopt: %s", strerror(errno));
        return EXIT_FAILURE;
    }

    if (bind(server_socket_fd, (struct sockaddr*)&server.addr,
        sizeof(server.addr)) == -1) {
        LOG_ERROR("bind: %s", strerror(errno));
    }
}

```

```

        return EXIT_FAILURE;
    }

    if (listen(server_socket_fd, server.queue_capacity) == -1) {
        LOG_ERROR("listen: %s", strerror(errno));
        return EXIT_FAILURE;
    }

    struct thread_pool* thread_pool = create_thread_pool(
        server.num_worker_threads);
    if (thread_pool == NULL) {
        LOG_ERROR("create_thread_pool: %s", strerror(errno));
        close(server_socket_fd);
        return EXIT_FAILURE;
    }

    server.running = true;
    __sighandler_t old_int_handler = signal(SIGINT,
        interrupt_handler);
    __sighandler_t old_pipe_handler = signal(SIGPIPE, SIG_IGN);

    LOG_INFO("server started at %s:%d",
        inet_ntoa(server.addr.sin_addr),
        ntohs(server.addr.sin_port));

    int ret = EXIT_SUCCESS;

    fd_set fdset;
    struct timeval timeout;
    while (server.running)
    {
        FD_ZERO(&fdset);
        FD_SET(server_socket_fd, &fdset);

        timeout.tv_sec = 10;
        timeout.tv_usec = 0;

        int ret = select(server_socket_fd + 1, &fdset, NULL,
            NULL, &timeout);
        if (ret == -1)
            break;
    }

```



```

        else if (ret > 0)
        {
            int client_socket = accept(server_socket_fd, NULL,
                                      NULL);
            if (client_socket == -1)
            {
                LOG_ERROR("accept: %s", strerror(errno));
                ret = EXIT_FAILURE;
                break;
            }

            LOG_DEBUG("new client connected: socket=%d",
                    client_socket);

            // post request job for thread pool
            post_request_job((struct request_job){
                .client_socket = client_socket,
            });
        }
    }

    signal(SIGINT, old_int_handler);
    signal(SIGPIPE, old_pipe_handler);
    close(server_socket_fd);
    free_all_request_jobs();
    destroy_thread_pool(thread_pool);
    return ret;
}

```

На листинге 3.4 представлен код работы с логгером.

Листинг 3.4 – Код работы с логгером

```

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "logger.h"

static log_level_t logger_level = LOG_LEVEL_INFO;
static log_redirect_t logger_redirect = LOG_REDIRECT_STDERR;
static FILE* logger_file = NULL;

int set_logger_file(const char* filename)

```

```

{
    if (logger_file)
        fclose(logger_file);
    logger_file = fopen(filename, "a");
    if (!logger_file)
    {
        perror("open");
        return EXIT_FAILURE;
    }
    setbuf(logger_file, NULL);
    return EXIT_SUCCESS;
}

void set_logger_level(log_level_t level)
{
    logger_level = level;
}

void set_logger_redirect(log_redirect_t redirect)
{
    logger_redirect = redirect;
}

void close_logger(void)
{
    if (logger_file)
    {
        fclose(logger_file);
        logger_file = NULL;
    }
}

void log_message(log_level_t level, const char* fmt, ...)
{
    if (level > logger_level)
        return;

    if (logger_redirect & LOG_REDIRECT_STDOUT)
    {
        va_list args;
        va_start(args, fmt);
    }
}

```

```

        vprintf(fmt, args);
        va_end(args);
    }

    if (logger_redirect & LOG_REDIRECT_STDERR)
    {
        va_list args;
        va_start(args, fmt);
        vfprintf(stderr, fmt, args);
        va_end(args);
    }

    if (logger_file && (logger_redirect & LOG_REDIRECT_FILE))
    {
        va_list args;
        va_start(args, fmt);
        vfprintf(logger_file, fmt, args);
        va_end(args);
    }
}

```

На листинге 3.5 представлен код работы с очередью запросов.

Листинг 3.5 – Код работы с очередью запросов

```

#include <errno.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "logger.h"
#include "request_queue.h"

struct job_list_node
{
    struct request_job job;
    struct job_list_node* next;
};

static bool queue_destroyed = false;
static struct job_list_node* queue = NULL;

```

```

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

inline static void lock(void)
{
    if (pthread_mutex_lock(&mutex) != 0)
    {
        LOG_ERROR("pthread_mutex_lock: %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

inline static void unlock(void)
{
    if (pthread_mutex_unlock(&mutex) != 0)
    {
        LOG_ERROR("pthread_mutex_unlock: %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

inline static void wakeup(void)
{
    if (pthread_cond_signal(&cond) != 0)
    {
        LOG_ERROR("pthread_cond_signal: %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

inline static void wakeup_all(void)
{
    if (pthread_cond_broadcast(&cond) != 0)
    {
        LOG_ERROR("pthread_cond_broadcast: %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

inline static void wait(void)
{

```

```

    if (pthread_cond_wait(&cond, &mutex) != 0)
    {
        LOG_ERROR("pthread_cond_wait: %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

int post_request_job(struct request_job job)
{
    struct job_list_node* node = calloc(1, sizeof(struct
        job_list_node));
    if (node == NULL)
    {
        LOG_ERROR("calloc: %s", strerror(errno));
        return EXIT_FAILURE;
    }
    node->job = job;

    // append job
    lock();
    struct job_list_node** last = &queue;
    while (*last != NULL)
        last = &(*last)->next;
    *last = node;
    wakeup();
    unlock();

    LOG_DEBUG("job posted client_socket=%d", job.client_socket);
    return EXIT_SUCCESS;
}

void free_all_request_jobs(void)
{
    lock();
    while (queue != NULL)
    {
        struct job_list_node* next = queue->next;
        free(queue);
        queue = next;
    }
    queue_destroyed = true;
}

```

```

    wakeup_all();
    unlock();

    LOG_DEBUG("request jobs queue destroyed");
}

int get_request_job(struct request_job* job)
{
    int ret = EXIT_SUCCESS;

    lock();
    while (queue == NULL && !queue_destroyed)
        wait();
    if (queue == NULL)
        ret = EXIT_FAILURE;
    else
    {
        *job = queue->job;
        struct job_list_node* next = queue->next;
        free(queue);
        queue = next;
    }
    unlock();
    return ret;
}

```

## 4 Исследовательский раздел

### 4.1 Анализ производительности программного обеспечения

Сравнение производительности разработанного веб-сервера с nginx проводится на основе 300 запросов, параллельно посылаются 10 запросов на один и тот же адрес и загружается pdf файл размером 173 Мбайт.

Для сравнительного анализа используется сервер nginx с конфигурацией, описанной в листинге 4.1. Результаты работы сервера оцениваются с помощью утилиты Apache Benchmark.

Листинг 4.1 – Конфигурация nginx

```
user root;

events {
    worker_connections 1024;
}

http {
    server {
        listen 8080;
        server_name localhost;

        location / {
            root /var/www;
            index index.html;
        }
    }
}
```

На рисунке 4.1 представлен результат анализа производительности.

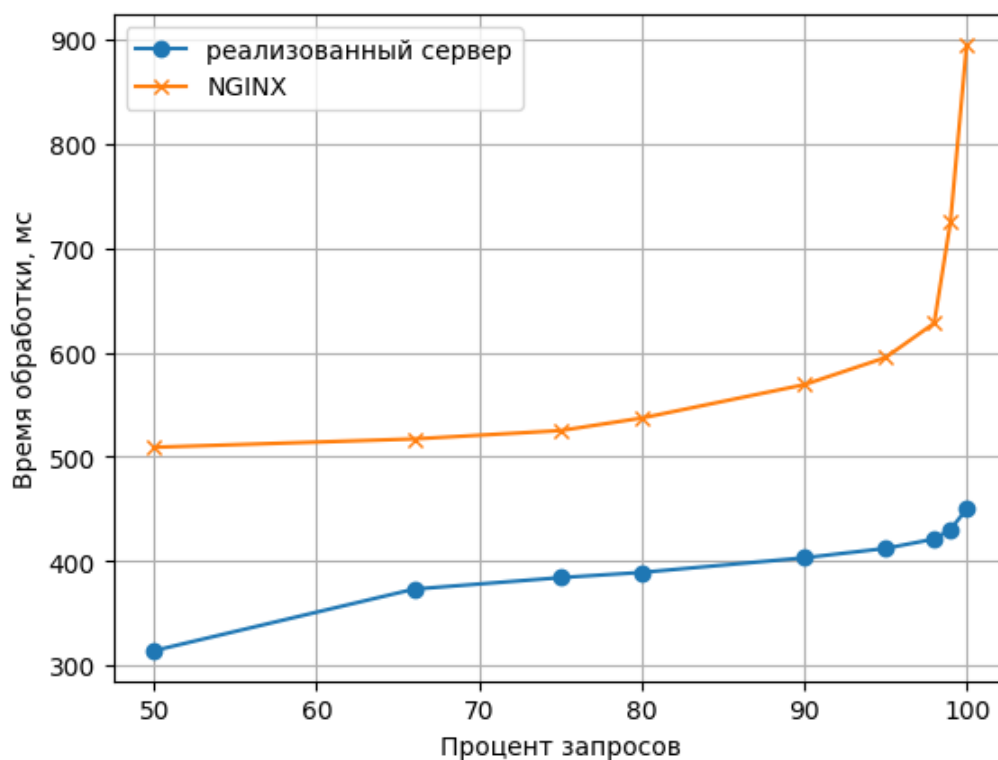


Рисунок 4.1 – Результаты анализа производительности

На основании результатов анализа можно сделать вывод о том, что разработанный веб-сервер эффективнее nginx, это может быть связано с тем, что во время тестирования сервера nginx были использованы механизмы контейнеризации, что создает дополнительные накладные расходы на сетевое взаимодействие.



## ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта была достигнута его цель — разработан сервер для отдачи статики с использованием механизмов, предоставляемых ядром операционной системы Linux.

Были решены следующие задачи:

- проведен анализ предметной области;
- спроектирован и реализован сервер;
- проведено сравнение разработанного сервера с nginx.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Yeager N. J., McGrath R. E.* Web server technology. — Morgan Kaufmann, 1996.
2. *Touch J., Heidemann J., Obraczka K.* Analysis of HTTP performance // URL: <http://www.isi.edu/touch/pubs/http-perf96/>, ISI Research Report ISI/RR-98-463,(original report dated Aug. 1996), USC/Information Sciences Institute. — 1998.
3. *Putro Z. P., Supono R. A.* Comparison Analysis of Apache and Nginx Webserver Load Balancing on Proxmox VE in Supporting Server Performance // International Research Journal of Advanced Engineering and Science. — 2022. — Т. 7, № 3. — С. 144—151.
4. *Tian Y.-C., Gao J.* Building TCP/IP Socket Applications // Network Analysis and Architecture. — Springer, 2023. — С. 501—540.
5. *Вильям С.* UNIX: Взаимодействие Процессов. — М.: Питер, 2003.
6. *Hunt C.* TCP/IP network administration. Т. 2. — "O'Reilly Media, Inc.", 2002.
7. *Gourley D., Totty B.* HTTP: the definitive guide. — "O'Reilly Media, Inc.", 2002.

## ПРИЛОЖЕНИЕ А