



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 сервера для отдачи статического
содержимого с диска»*

Студент ИУ7-71Б
(Группа)

(Подпись, дата)

Постнов С. А.
(Фамилия И. О.)

Руководитель курсовой работы

(Подпись, дата)

Клочков М. Н.
(Фамилия И. О.)

2024 г.

РЕФЕРАТ

Расчетно-пояснительная записка 25 с., 9 рис., 1 табл., 7 источн., 1 прил.
ВЕБ-СЕРВЕР, THREAD-POOL, PSELECT, NGINX, C.

Цель работы — разработка сервера для отдачи статического содержимого с диска. Архитектура сервера должна быть основана на пуле потоков (thread-pool) совместно с `pselect()`.

В результате работы был проведен анализ предметной области, спроектирована схема алгоритма работы веб-сервера. Разработан статический веб-сервер, поддерживающий обработку запросов на получение различных типов файлов. Проведено сравнение реализованного веб-сервера с существующими аналогами.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 Аналитический раздел	6
1.1 Архитектура thread-pool	6
1.2 Системный вызов pselect	7
2 Конструкторский раздел	9
2.1 Схема алгоритма работы сервера	9
3 Технологический раздел	10
3.1 Требования к разрабатываемой программе	10
3.2 Реализация веб-сервера	11
3.3 Примеры работы программы	18
4 Исследовательский раздел	21
4.1 Технические характеристики	21
4.2 Описание исследования	21
4.3 Результаты исследования	22
ЗАКЛЮЧЕНИЕ	23
ПРИЛОЖЕНИЕ А	24
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	25

ВВЕДЕНИЕ

Статический веб-сервер — сервер, который обслуживает статические файлы по запросу клиента. Статические файлы — файлы, содержимое которых не изменяется динамически на стороне сервера. Статический сервер предназначен для чтения файлов из диска и отправления их клиенту, как правило, по протоколу HTTP.

Целью курсовой работы является разработка сервера для отдачи статического содержимого с диска. Архитектура сервера должна быть основана на пуле потоков (thread-pool) совместно с `pselect()`.

Для достижения поставленной цели необходимо решить следующие задачи:

- 1) описать предметную область;
- 2) спроектировать схему алгоритма работы сервера;
- 3) выбрать средства реализации сервера;
- 4) провести сравнение реализованного сервера с известными аналогами.

1 Аналитический раздел

1.1 Архитектура thread-pool

Архитектура пул потоков (thread-pool) — модель управления потоками выполнения (threads), которая предусматривает предварительное создание фиксированного или динамически изменяемого пула потоков, используемых для обработки задач из общей очереди. Такая архитектура обеспечивает эффективное распределение вычислительных ресурсов и уменьшение накладных расходов, связанных с созданием и уничтожением потоков. Она является одной из самых распространенных архитектур многопоточности Object Request Broker Architecture (CORBA), используемых в реализациях Object Request Broker (ORB), и была принята веб-серверами, такими как Microsoft Internet Information Server (IIS) [1].

Основными компонентами архитектуры thread-pool являются:

- 1) пул потоков (thread-pool);
- 2) очередь задач (task queue);
- 3) менеджер пула (pool manager);
- 4) задачи (tasks).

Архитектура пула потоков представлена на рисунке 1.1.



Рисунок 1.1 – Архитектура пула потоков

1.2 Системный вызов pselect

В Unix процесс выполняет ввод-вывод по одному файловому дескриптору за раз, поэтому происходит блокировка и снижение производительности программы. Чтобы избежать этой проблемы, необходимо использовать системный вызов **pselect()**, который позволяет программе отслеживать несколько файловых дескрипторов. Программа ожидает, пока один или несколько файловых дескрипторов не станут готовы к определённому классу операций ввода-вывода, не блокируя их и обеспечивая многопоточный синхронный ввод-вывод [2].

Модель неблокирующего синхронного ввода-вывода, в которой применяется `pselect()`, представлена на рисунке 1.2.

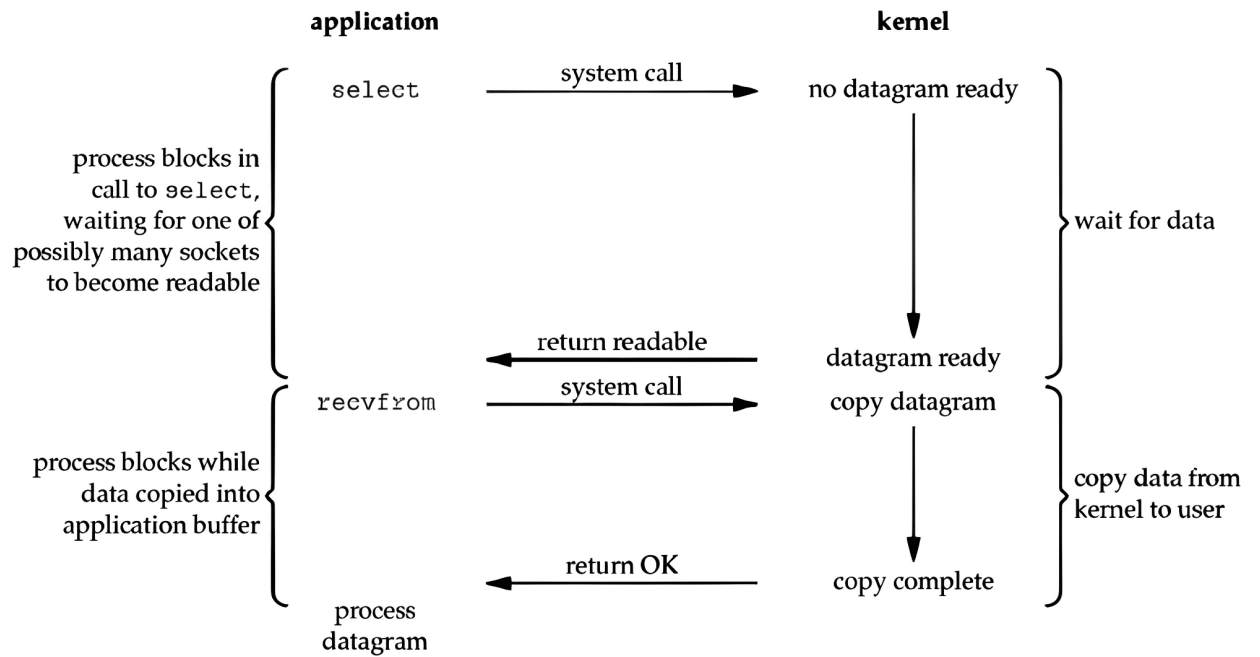


Рисунок 1.2 – Модель неблокирующего синхронного ввода-вывода

2 Конструкторский раздел

2.1 Схема алгоритма работы сервера

На рисунке 2.1 представлена схема алгоритма работы сервера.

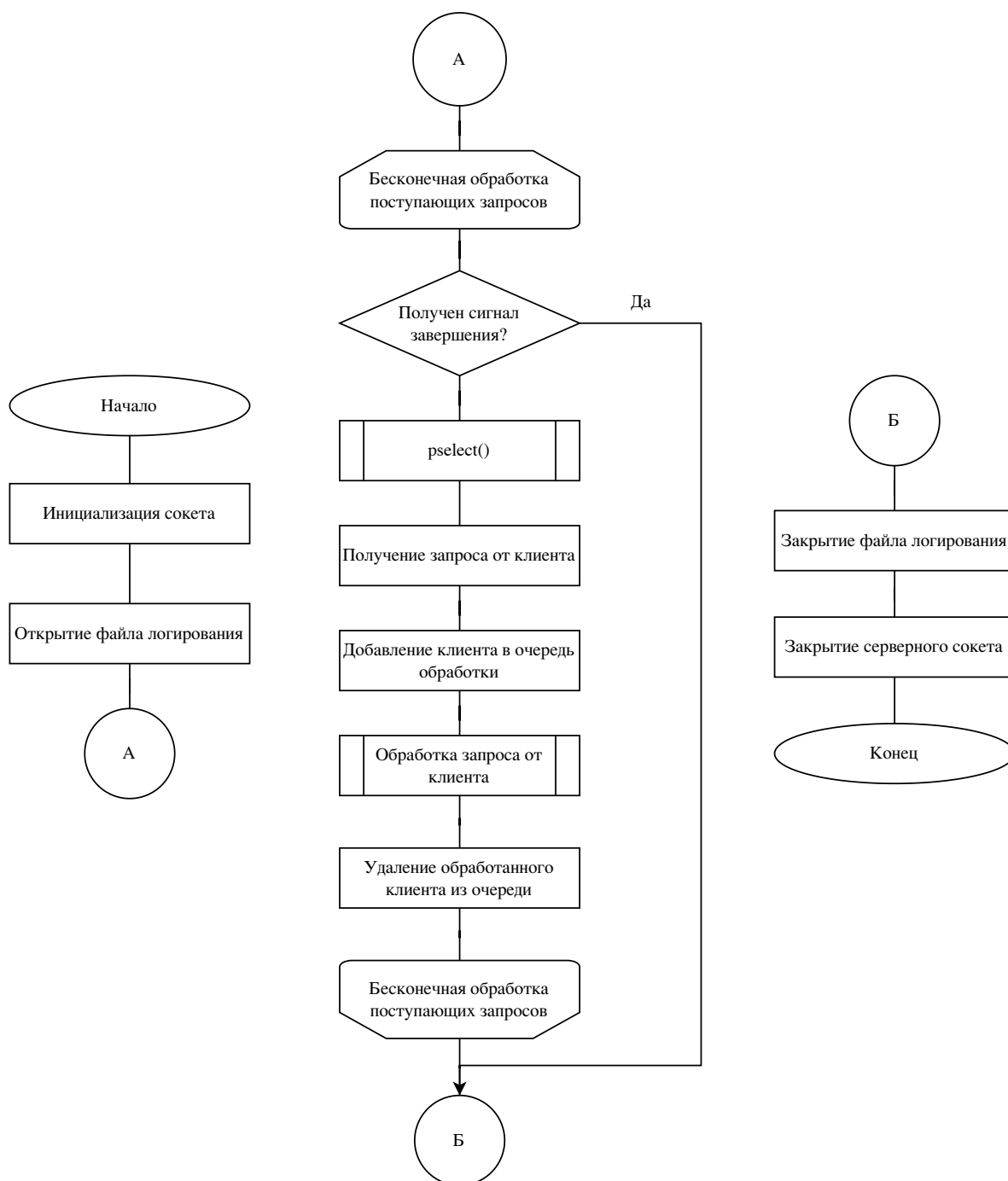


Рисунок 2.1 – Схема алгоритма работы сервера

3 Технологический раздел

3.1 Требования к разрабатываемой программе

Разрабатываемое программное обеспечение должно удовлетворять следующим требованиям:

- 1) поддержка запросов GET и HEAD;
- 2) поддержка статусов 200, 403, 404 и 405 (на неподдерживаемые запросы);
- 3) поддержка корректной передачи файлов размером до 128 Мб;
- 4) возврат по умолчанию html-страницы с css-стилем;
- 5) запись информации о событиях;
- 6) минимальные требования к безопасности серверов статического содержимого.

В качестве языка программирования для реализации веб-сервера был выбран язык C. В качестве среды для разработки была выбрана среда CLion [3; 4].

3.2 Реализация веб-сервера

Реализация веб-сервера представлена в листинге 3.1.

Листинг 3.1 – Реализация веб-сервера

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/stat.h>
#include <signal.h>
#include <time.h>

#define PORT 8080
#define MAX_THREADS 11
#define QUEUE_SIZE 64
#define ROOT_DIR "./static"
#define MAX_BUFFER 4096
#define LOG_BUFFER 512

pthread_mutex_t queue_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t queue_cond = PTHREAD_COND_INITIALIZER;

int task_queue[QUEUE_SIZE];
int queue_start = 0,
    queue_end = 0,
    queue_count = 0;

FILE *log_file = NULL;
int server_fd;

int endswith(const char *str, const char *suffix) {
    if (!str || !suffix)
        return 0;

    const size_t str_len = strlen(str);
```

```

    const size_t suffix_len = strlen(suffix);
    if (suffix_len > str_len)
        return 0;

    return strncmp(
        str + str_len - suffix_len,
        suffix,
        suffix_len) == 0;
}

void log_event(const char *message) {
    pthread_mutex_lock(&queue_mutex);
    if (log_file) {
        const time_t now = time(NULL);
        fprintf(log_file, "[%s] %s\n",
            strtok(ctime(&now), "\n"),
            message);
        fflush(log_file);
    }
    pthread_mutex_unlock(&queue_mutex);
}

void cleanup() {
    if (log_file)
        fclose(log_file);
    if (server_fd > 0)
        close(server_fd);

    pthread_mutex_destroy(&queue_mutex);
    pthread_cond_destroy(&queue_cond);
}

void handle_signal(const int sig) {
    if (sig == SIGINT || sig == SIGKILL) {
        log_event("Server shutting down...");
        cleanup();
        exit(0);
    }
}

void enqueue_task(const int fd) {

```

```

pthread_mutex_lock(&queue_mutex);
if (queue_count == QUEUE_SIZE) {
    pthread_mutex_unlock(&queue_mutex);
    close(fd);
    return;
}

task_queue[queue_end] = fd;
queue_end = (queue_end + 1) % QUEUE_SIZE;
queue_count++;
pthread_cond_signal(&queue_cond);
pthread_mutex_unlock(&queue_mutex);
}

int dequeue_task() {
    pthread_mutex_lock(&queue_mutex);
    while (queue_count == 0)
        pthread_cond_wait(
            &queue_cond,
            &queue_mutex);

    const int fd = task_queue[queue_start];
    queue_start = (queue_start + 1) % QUEUE_SIZE;
    --queue_count;
    pthread_mutex_unlock(&queue_mutex);

    return fd;
}

void send_response(const int fd,
                  const int status,
                  const char *status_text,
                  const char *content_type,
                  const char *body,
                  const size_t body_length) {
    char header[MAX_BUFFER];
    const int header_length = snprintf(header, MAX_BUFFER,
        "HTTP/1.1 %d %s\r\n"
        "Content-Length: %zu\r\n"
        "Content-Type: %s\r\n"
        "Connection: close\r\n\r\n",

```

```

        status, status_text, body_length, content_type);

write(fd, header, header_length);
if (body && body_length > 0)
    write(fd, body, body_length);
close(fd);

char log_msg[LOG_BUFFER];
snprintf(log_msg, sizeof(log_msg),
         "Response: %d %s", status, status_text);
log_event(log_msg);
}

void serve_static_file(const int fd, const char *file_path) {
    struct stat file_stat;
    if (stat(file_path, &file_stat) == -1 ||
        S_ISDIR(file_stat.st_mode)) {
        send_response(fd, 404, "Not Found",
                     "text/html",
                     "<h1>404 Not Found</h1>", 22);
        return;
    }

    if (access(file_path, R_OK) == -1) {
        send_response(fd, 403, "Forbidden",
                     "text/html",
                     "<h1>403 Forbidden</h1>", 22);
        return;
    }

    const int file_fd = open(file_path, O_RDONLY);
    if (file_fd == -1) {
        send_response(fd, 500, "Internal Server Error",
                     "text/html",
                     "<h1>500 Internal Error</h1>", 27);
        return;
    }

    char header[MAX_BUFFER];
    snprintf(header, sizeof(header),
             "HTTP/1.1 200 OK\r\nContent-Length: %ld"

```

```

        "\r\nContent-Type: %s\r\n"
        "Connection: close\r\n\r\n",
file_stat.st_size,
(endswith(file_path, ".html") ? "text/html" :
                                     "text/plain"));
write(fd, header, strlen(header));

ssize_t bytes_read;
char buffer[MAX_BUFFER];
while ((bytes_read = read(
        file_fd,
        buffer,
        sizeof(buffer))) > 0)
    write(fd, buffer, bytes_read);

close(file_fd);
close(fd);

log_event("Response: 200 OK");
}

void *worker_thread(void *arg) {
    while (1) {
        const int client_fd = dequeue_task();

        char buffer[MAX_BUFFER];
        const ssize_t bytes_read = read(
            client_fd,
            buffer,
            sizeof(buffer) - 1);
        if (bytes_read <= 0) {
            close(client_fd);
            continue;
        }
        buffer[bytes_read] = '\0';

        char method[16], path[256];
        sscanf(buffer, "%15s %255s", method, path);

        char log_msg[LOG_BUFFER];
        snprintf(log_msg, sizeof(log_msg),

```

```

        "Request: %s on %s", method, path);
log_event(log_msg);

if (strcmp(method, "GET") != 0 &&
    strcmp(method, "HEAD") != 0) {
    send_response(client_fd, 405,
                  "Method Not Allowed",
                  "text/html",
                  "<h1>405 Method Not Allowed</h1>",
                  31);
    continue;
}

if (strlen(path) == 1 && path[0] == '/')
    strncpy(path, "/index.html", 11);

char file_path[512];
snprintf(file_path, sizeof(file_path), "%s%s",
         ROOT_DIR, path);
serve_static_file(client_fd, file_path);
}
}

int main() {
    signal(SIGINT, handle_signal);
    log_file = fopen("server.log", "a");
    if (!log_file) {
        perror("Failed to open log file");
        return 1;
    }

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("Socket creation failed");
        return 1;
    }

    struct sockaddr_in server_addr = {0};
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

```

```

if (bind(server_fd, (struct sockaddr *)&server_addr,
        sizeof(server_addr)) == -1) {
    perror("Bind failed");
    return 1;
}

if (listen(server_fd, SOMAXCONN) == -1) {
    perror("Listen failed");
    return 1;
}

char log_msg[LOG_BUFFER];
snprintf(log_msg, sizeof(log_msg),
        "Server starting on :%d", PORT);
log_event(log_msg);

pthread_t threads[MAX_THREADS];
for (int i = 0; i < MAX_THREADS; ++i)
    pthread_create(
        &threads[i],
        NULL,
        worker_thread,
        NULL);

fd_set readfds;
FD_ZERO(&readfds);
FD_SET(server_fd, &readfds);
const int max_fd = server_fd;

while (1) {
    fd_set temp_set = readfds;
    if (pselect(max_fd + 1, &temp_set,
        NULL, NULL, NULL, NULL) > 0) {
        if (FD_ISSET(server_fd, &temp_set)) {
            const int client_fd = accept(
                server_fd,
                NULL,
                NULL);
            if (client_fd == -1) {
                perror("Accept failed");
            }
        }
    }
}

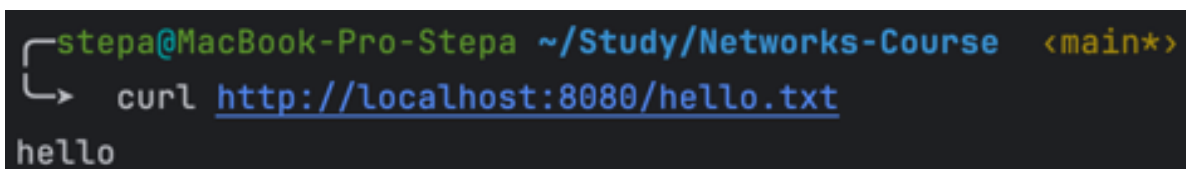
```



```
        continue;
    }
    enqueue_task(client_fd);
}
}
}
```

3.3 Примеры работы программы

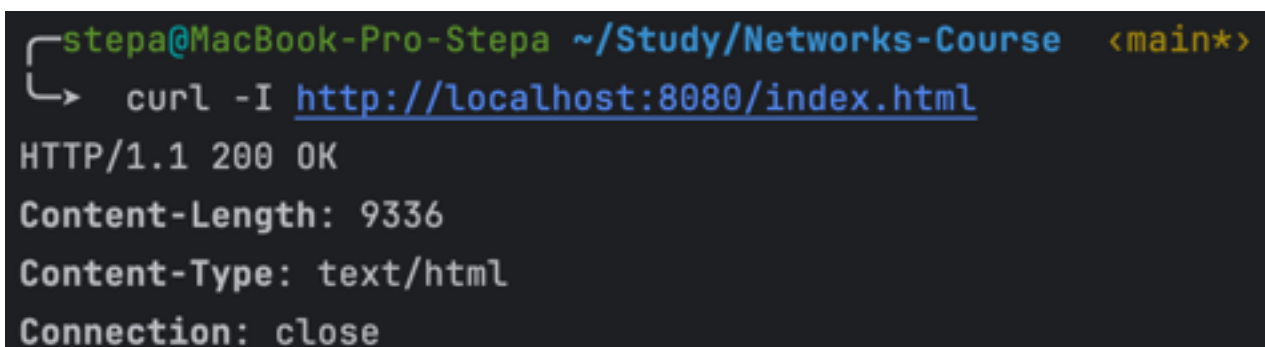
На рисунке 3.1 представлен пример ответа на GET-запрос.



```
stepa@MacBook-Pro-Stepa ~/Study/Networks-Course <main*>
└─> curl http://localhost:8080/hello.txt
hello
```

Рисунок 3.1 – Пример ответа на GET-запрос

На рисунке 3.2 представлен пример ответа на HEAD-запрос.



```
stepa@MacBook-Pro-Stepa ~/Study/Networks-Course <main*>
└─> curl -I http://localhost:8080/index.html
HTTP/1.1 200 OK
Content-Length: 9336
Content-Type: text/html
Connection: close
```

Рисунок 3.2 – Пример ответа на HEAD-запрос

На рисунке 3.3 представлен пример ответа на GET-запрос несуществующего файла.

```
stepa@MacBook-Pro-Stepa ~/Study/Networks-Course <main*>
└─ curl -I http://localhost:8080/indexxx.html
HTTP/1.1 404 Not Found
Content-Length: 22
Content-Type: text/html
Connection: close
```

Рисунок 3.3 – Пример ответа на GET-запрос несуществующего файла

На рисунке 3.4 представлен пример ответа на неразрешенный POST-запрос.

```
stepa@MacBook-Pro-Stepa ~/Study/Networks-Course <main*>
└─ curl -X POST http://localhost:8080/1.png
<h1>405 Method Not Allowed</h1>%
```

Рисунок 3.4 – Пример ответа на неразрешенный POST-запрос

На рисунке 3.5 представлен пример ответа на GET-запрос без прав на доступ.

```
stepa@MacBook-Pro-Stepa ~/Study/Networks-Course <main*>
└─ chmod 0 static/hello.txt

stepa@MacBook-Pro-Stepa ~/Study/Networks-Course <main*>
└─ curl http://localhost:8080/hello.txt
<h1>403 Forbidden</h1>%
```

Рисунок 3.5 – Пример ответа на GET-запрос без прав на доступ

На рисунке 3.6 представлен пример логирования событий в системе.

```
stepa@MacBook-Pro-Stepa ~/Study/Networks-Course <main*>
└─ cat server.log
[Sun Dec 8 20:39:45 2024] Server starting on :8080
[Sun Dec 8 20:39:59 2024] Request: GET on /
[Sun Dec 8 20:39:59 2024] Response: 200 OK
[Sun Dec 8 20:40:19 2024] Request: GET on /
[Sun Dec 8 20:40:19 2024] Response: 200 OK
[Sun Dec 8 20:40:23 2024] Request: GET on /hello.txt
[Sun Dec 8 20:40:23 2024] Response: 200 OK
[Sun Dec 8 20:40:41 2024] Request: GET on /hello.txt
[Sun Dec 8 20:40:41 2024] Response: 200 OK
[Sun Dec 8 20:40:49 2024] Request: GET on /hello.txt
[Sun Dec 8 20:40:49 2024] Response: 200 OK
[Sun Dec 8 20:41:23 2024] Request: HEAD on /1.png
[Sun Dec 8 20:41:23 2024] Response: 200 OK
[Sun Dec 8 20:41:32 2024] Request: HEAD on /index.html
[Sun Dec 8 20:41:32 2024] Response: 200 OK
[Sun Dec 8 20:42:03 2024] Request: HEAD on /indexxx.html
[Sun Dec 8 20:42:03 2024] Response: 404 Not Found
```

Рисунок 3.6 – Пример логирования событий в системе

4 Исследовательский раздел

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- 1) операционная система — macOS 15.1.1 (24B91) [5];
- 2) объем оперативной памяти — 18 Гбайт;
- 3) процессор — Apple M3 Pro, 11 ядер [5].

Во время тестирования ноутбук был подключен к сети электропитания и нагружен только встроенными приложениями и системой тестирования.

4.2 Описание исследования

В качестве стороннего веб-сервера для сравнения производительности был выбран **nginx**, так как является самым популярным и востребованным [6]. В качестве инструмента для генерации нагрузки и замеров производительности будет использоваться утилита **ab** (Apache Benchmark) [7]. Целью исследования является сравнение реализованного веб-сервера с известными аналогами по среднему времени ответа на получение файла размером 9.1 КБ в зависимости от количества запросов.

4.3 Результаты исследования

Результаты сравнения веб-серверов представлены в таблице 4.1.

Таблица 4.1 – Среднее время ответа на запрос для получения файла

Количество запросов	Среднее время ответа (разработанный веб-сервер), мс	Среднее время ответа (nginx), мс
100	0.209	0.739
1000	0.091	0.460
5000	0.079	0.440
10000	0.077	0.444
50000	0.077	0.444
100000	0.077	0.444

Вывод

В исследовательском разделе было проведено сравнение реализованного статического веб-сервера и веб-сервера **nginx** по времени получения файла размером 9.1 КБ. Исходя из полученных в таблице 4.1 результатов, был сделан вывод, что время ответа для небольшого количества запросов (до 1000) примерно в 2 раза больше, чем для большого количества запросов (больше 1000). При этом разница во времени ответа между разным количеством запросов уменьшается с повышением самого числа запросов для обоих серверов. Реализованный веб-сервер в среднем превосходит **nginx** по скорости ответа примерно в 3.5 раза до 1000 запросов и примерно в 6 раз для количества запросов, превышающего 1000.

ЗАКЛЮЧЕНИЕ

Цель работы, заключающаяся в разработке сервера для отдачи статического содержимого с диска с использованием архитектуры, основанной на пуле потоков (thread-pool) совместно с `pselect()`, была достигнута.

Были решены следующие задачи:

- 1) описана предметная область;
- 2) спроектирована схема алгоритма работы сервера;
- 3) выбраны средства реализации сервера;
- 4) проведено сравнение реализованного сервера с известными аналогами.

ПРИЛОЖЕНИЕ А

Презентация к курсовой работе состоит из 9 слайдов

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Yibei Ling, Tracy Mullen, Xiaola Lin*. Analysis of Optimal Thread Pool Size. — 2000.
2. MAN pselect. — [Электронный ресурс]. — Режим доступа: <https://www.opennet.ru/man.shtml?topic=pselect&category=2&russian=0> (дата обращения: 23.11.24).
3. C Language Reference. — [Электронный ресурс]. — Режим доступа: <https://learn.microsoft.com/en-us/cpp/c-language/c-language-reference?view=msvc-170> (дата обращения: 24.11.24).
4. A cross-platform IDE for C and C++. — [Электронный ресурс]. — Режим доступа: <https://www.jetbrains.com/clion/> (дата обращения: 24.11.24).
5. MacOS. — [Электронный ресурс]. — Режим доступа: <https://www.apple.com/macos/macos-sequoia/> (дата обращения: 25.11.24).
6. NGINX. — [Электронный ресурс]. — Режим доступа: <https://nginx.org/ru/> (дата обращения: 26.11.24).
7. Apache Benchmark. — [Электронный ресурс]. — Режим доступа: <https://httpd.apache.org/docs/current/programs/ab.html> (дата обращения: 26.11.24).