РЕФЕРАТ

Расчетно-пояснительная записка 44 с., 7 рис., 2 таблицы, 16 источников, 0 приложений.

СТАТИЧЕСКИЙ СЕРВЕР, ВЕБ-СЕРВЕР, HTTP, NGINX, НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ, APACHE BENCHMARK.

Цель работы — реализация классического статического веб-сервера для отдачи контента с диска.

Была изучена предметная область, связанная со статическим веб-сервером. Был проведен анализ протокола HTTP. Была осуществлена формализация бизнес-правил разрабатываемого программного обеспечения.

Были сформулированы требования к разрабатываемому статическому веб-серверу. Были проанализированы сокеты как средство взаимодействия между процессами. Были разработаны схемы алгоритмов работы статического веб-сервера и обработки HTTP-запросов.

Были выбраны средства реализации статического веб-сервера. Был написан код программного обеспечения для отдачи контента с диска по HTTP-запросу.

Было проведено сравнение результатов нагрузочного тестирования разработанного программного обеспечения и сервера, развернутого на базе nginx. Согласно полученным данным, сервер на базе nginx выполняет запросы в среднем в 2.5 раза быстрее, чем разработанное программное обеспечение.

Содержание

ВВЕДЕНИЕ

Целью работы является реализация классического статического веб-сервера для отдачи контента с диска.

Задачи работы:

— анализ предметной области, связанной со статическим веб-сервером;

— предъявление требований к разрабатываемому программному обеспечению;

— проектирование архитектуры статического веб-сервера для отдачи контента с диска;

— реализация статического веб-сервера для отдачи контента с диска;

— исследование характеристик реализованного сервера.

1 Аналитическая часть

1.1 Статический веб-сервер

На самом базовом уровне, когда браузеру нужен файл, размещённый на веб-сервере, браузер запрашивает его через HTTP-протокол [1]. Когда запрос достигает нужного веб-сервера, HTTP-сервер принимает запрос, находит запрашиваемый документ и отправляет обратно также через HTTP [1].

Схема взаимодействия браузера и веб-сервера представлена на рисунке 1.1. Статический веб-сервер посылает состоит из компьютера с HTTP-сервером и посылает размещенные в нем файлы в браузер без изменения их содержимого [1].

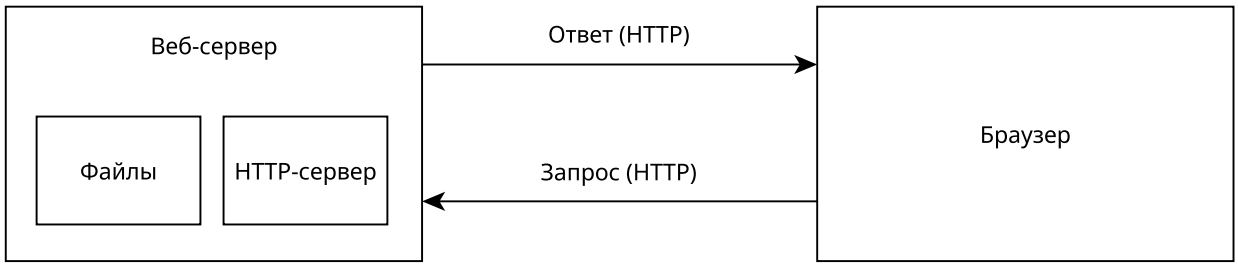

Рисунок 1.1 — Схема взаимодействия браузера и веб-сервера

1.2 Протокол HTTP

HTTP (Hypertext Transfer Protocol) — протокол прикладного уровня для передачи данных между узлами распределённых, объединённых, гипермедийных информационных систем [2]. В основе HTTP лежит концепция «клиент-сервер», то есть предполагается существование процессов-потребителей, которые инициируют соединение и посылают запрос, и процессов-поставщиков, которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом [2].

Протокол HTTP определяет два типа сообщений: запросы и ответы [2]. Каждое HTTP-сообщение состоит из следующих элементов [2]:

— стартовая строка,

— набор заголовков,

— тело.

Структура стартовой строки зависит от типа сообщения [2]. В HTTP/1.1 стартовая строка запроса имеет следующий формат [2]:

<VERB> <URI> HTTP/1.1,

где VERB — тип запроса;

URI — идентификатор ресурса.

Стартовая строка ответа в HTTP/1.1 имеет следующую структуру [2]:

HTTP/1.1 <CODE> <DESC>,

где CODE — код состояния ответа;

DESC — пояснение к коду ответа.

Код состояния ответа HTTP показывает, был ли успешно выполнен HTTP-запрос [2]. В HTTP/1.1 приводится следующая классификация кодов состояния [2]:

— информационные (100 – 199);

— успешные (200 – 299);

— перенаправления (300 – 399);

— клиентские ошибки (400 – 499);

— серверные ошибки (500 – 599).

Тип запроса (он же метод или глагол) определяет тип манипуляции над данными [2]. В HTTP/1.1 существуют следующие глаголы [2]:

— OPTIONS (описание параметров для соединения с ресурсом);

— GET (запрос представления данных);

— HEAD (запрос представления данных без тела ответа);

— POST (отправка сущностей к определенному ресурсу);

— PUT (замена всех текущих представлений ресурса данными запроса);

— DELETE (удаление указанного ресурса);

— TRACE (вызов возвращаемого тестового сообщения с ресурса);

— CONNECT (устанавливает «туннель» к серверу, определенному по ресурсу).

Заголовки HTTP позволяют клиенту и серверу отправлять дополнительную информацию с HTTP запросом или ответом [2]. В HTTP-заголовке содержится нечувствительное к регистру название, а затем после символа двоеточия — непосредственно значение [2]. Пробелы перед значением игнорируются [2].

1.3 Формализация бизнес-правил

На рисунке **??** представлена формализация бизнес-правил разрабатываемого программного обеспечения в нотации IDEF0.

Вывод

Была изучена предметная область, связанная со статическим веб-сервером. Был проведен анализ протокола HTTP. Была осуществлена формализация бизнес-правил разрабатываемого программного обеспечения.

2 Конструкторская часть

2.1 Требования к разрабатываемому программному обеспечению

К разрабатываемому статическому веб-серверу предъявляются следующие требования:

— работа сервера в операционной системе на базе ядра Linux;

— соответствие сервера архитектуре prefork + pselect;

— поддержка запросов GET и HEAD;

— поддержка кодов состояний 200, 403, 404 и 405;

— поддержка HTML-, CSS-, JS-, PNG-, JPG-, JPEG-, SWF- и GIF-файлов;

— соответствие минимальным требованиям к безопасности статических серверов (ошибка в случае выхода адреса за корень директории сервера);

— обеспечение логирования.

2.2 Сокеты

Сокеты — название программного интерфейса для обеспечения обмена данными между процессами [3]. Процессы при таком обмене могут исполняться как на одной электронной вычислительной машине, так и на различных компьютерах, связанных между собой сетью [3].

Каждый процесс может создать слушающий сокет и привязать его к определенному порту операционной системы [3]. Слушающий процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения [3]. При этом сохраняется возможность проверить наличие соединений на данный момент, установить тайм-аут для операции и так далее [3].

Каждый сокет имеет свой адрес [3]. ОС семейства UNIX могут поддерживать много типов адресов, но обязательными являются INET-адрес и UNIX-адрес [3]. Если привязать сокет к UNIX-адресу, то будет создан специальный файл по заданному пути, через который смогут общаться любые локальные процессы путем чтения или записи из него [3]. Сокеты типа INET доступны из сети и требуют выделения номера порта [3]. Обычно клиент явно подсоединяется к слушателю, после чего любое чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером [3]. Все сокеты обычно ориентированы на применение датаграмм, но их точные характеристики зависят от интерфейса, обеспечиваемого протоколом [3].

На рисунке 2.1 представлен процесс установления соединения и обмена данными между сокетами сервера и клиента в операционной системе Linux.

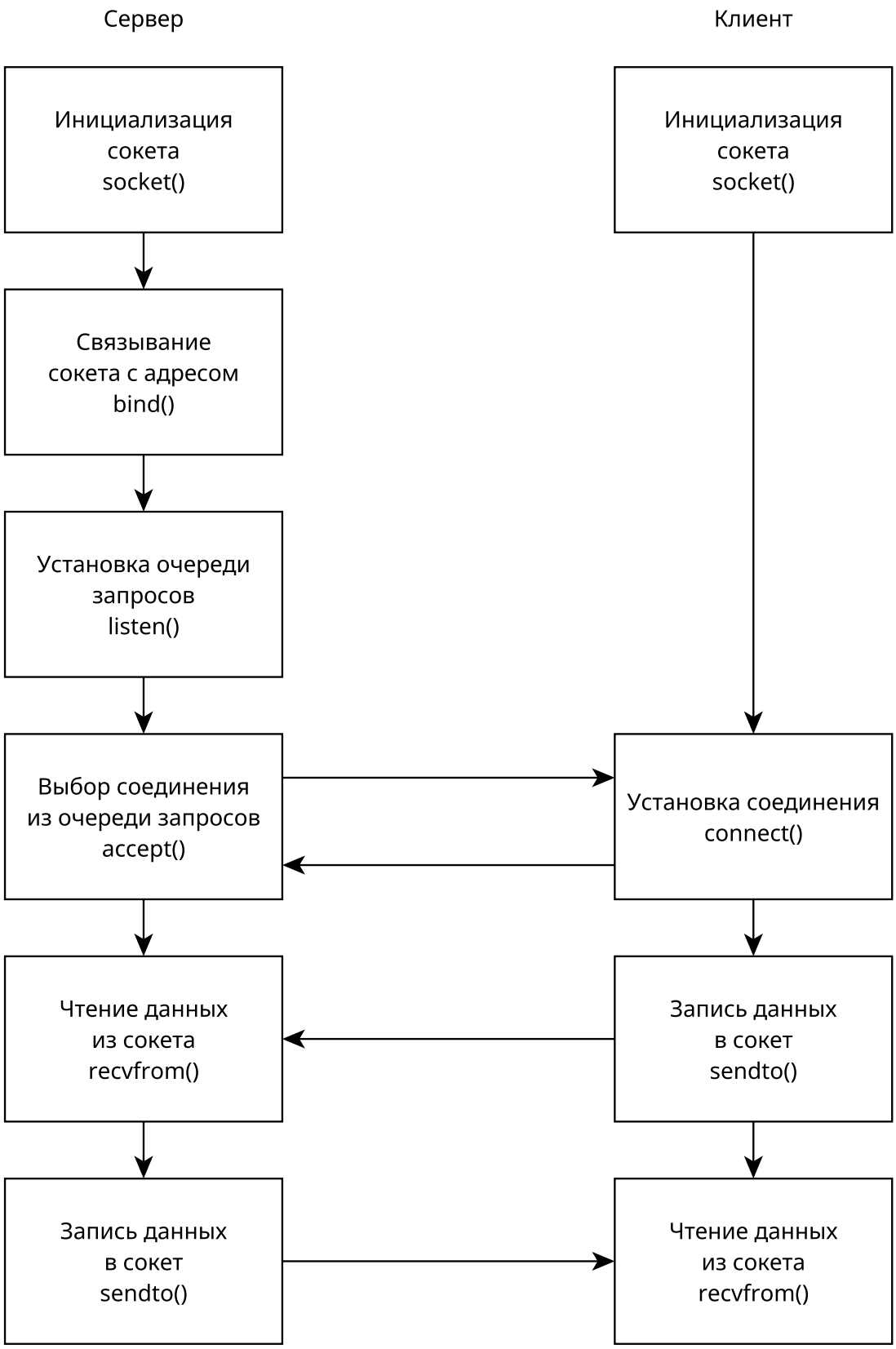

Рисунок 2.1 — Процесс установления соединения и обмена данными между сокетами сервера и клиента в операционной системе Linux

2.3 Разработка алгоритмов, необходимых для работы статического веб-сервера

На рисунке 2.2 представлена схема алгоритма работы статического сервера.

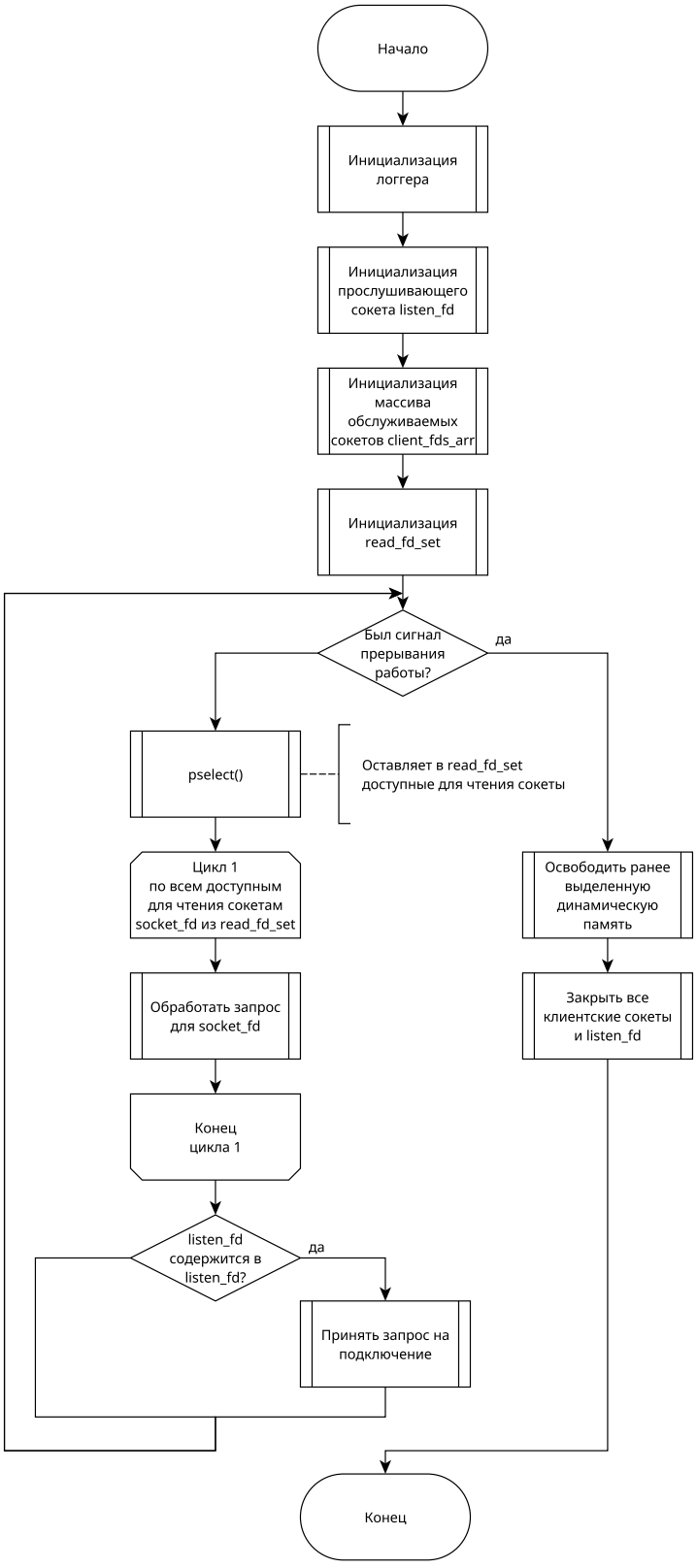

Рисунок 2.2 — Схема алгоритма работы статического сервера

На рисунке 2.3 представлена схема алгоритма обработки HTTP-запроса.

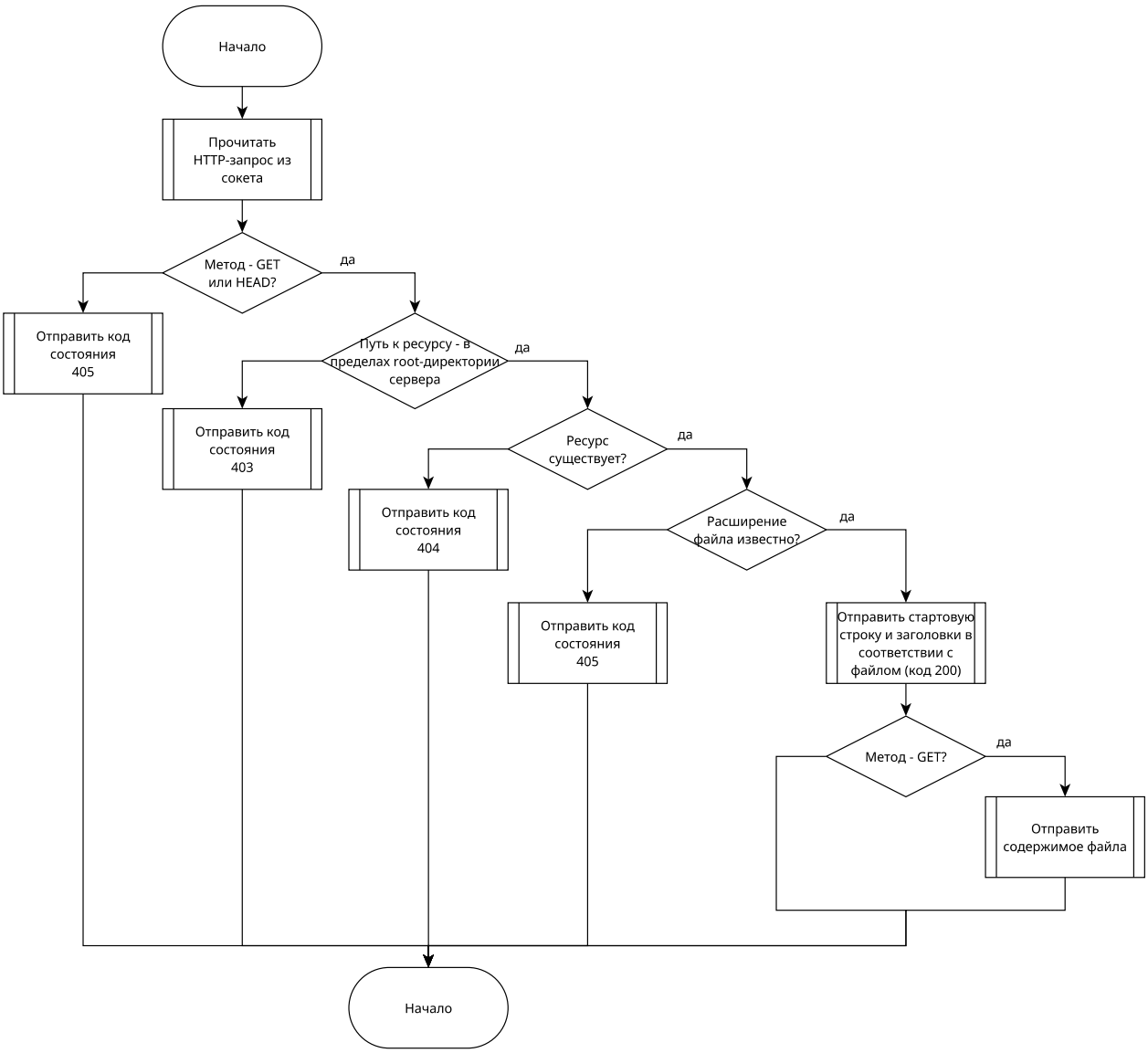

Рисунок 2.3 — Схема алгоритма обработки HTTP-запроса

Вывод

Были сформулированы требования к разрабатываемому статическому веб-серверу. Были проанализированы сокеты как средство взаимодействия между процессами. Были разработаны схемы алгоритмов работы статического веб-сервера и обработки HTTP-запросов.

3 Технологическая часть

Для реализации статического веб-сервера был использован язык программирования C. Программное обеспечение было разработано для операционных систем на базе ядра Linux. Для инициализации сервера были использованы системные вызовы socket, bind, listen и accept [4–7]. Для чтения запросов и записи ответов были применены функции recvfrom и sendto соответственно [8, 9]. Для реализации требуемой архитектуры были использованы системные вызовы fork и pselect [10, 11].

В листинге 3.1 представлена реализация логгера статического веб-сервера.

Листинг 3.1 — Реализация логгера статического веб-сервера

```
#include "logger.h"

int logger_init(
  logger_t *const logger_ptr,
  const char *log_file_name,
  const loglevel_t level
)
{
  logger_ptr->log_file_fd = open(log_file_name, O_WRONLY | O_CREAT,
     S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

  if (logger_ptr->log_file_fd == -1)
    return EXIT_FAILURE;

  logger_ptr->level = level;

  return EXIT_SUCCESS;
}

void logger_close(logger_t *const logger_ptr)
{
  close(logger_ptr->log_file_fd);
}

static void print_msg(
  const int fd,
  const char *msg,
  const char *prefix
)
{
  char log_msg[MAX_LOG_MSG_LENGTH] = {0};
```

```
  time_t cur_local_time = time(NULL);

  char *cur_local_time_str = asctime(localtime(&cur_local_time));
  cur_local_time_str[strlen(cur_local_time_str) - 1] = 0;

  sprintf(
    log_msg,
    "[%s] %s: %s\n",
    cur_local_time_str,
    prefix,
    msg
  );

  write(fd, log_msg, strlen(log_msg) + 1);
}

void logger_fatal(
  logger_t *const logger_ptr,
  const char *msg
)
{
  if (logger_ptr->level >= FATAL)
    print_msg(logger_ptr->log_file_fd, msg, "FATAL");
}

void logger_error(
  logger_t *const logger_ptr,
  const char *msg
)
{
  if (logger_ptr->level >= ERROR)
    print_msg(logger_ptr->log_file_fd, msg, "ERROR");
}

void logger_warn(
  logger_t *const logger_ptr,
  const char *msg
)
{
  if (logger_ptr->level >= WARN)
```

```
    print_msg(logger_ptr->log_file_fd, msg, "WARN");
}

void logger_info(
  logger_t *const logger_ptr,
  const char *msg
)
{
  if (logger_ptr->level >= INFO)
    print_msg(logger_ptr->log_file_fd, msg, "INFO");
}

void logger_debug(
  logger_t *const logger_ptr,
  const char *msg
)
{
  if (logger_ptr->level >= DEBUG)
    print_msg(logger_ptr->log_file_fd, msg, "DEBUG");
}

void logger_trace(
  logger_t *const logger_ptr,
  const char *msg
)
{
  if (logger_ptr->level >= TRACE)
    print_msg(logger_ptr->log_file_fd, msg, "TRACE");
}
```

В листинге 3.2 представлена реализация обработчика HTTP-запросов.

Листинг 3.2 — Реализация обработчика HTTP-запросов

```
#include "httptools.h"

#define REQ_LINE_PARTS_COUNT 2

#define READ_BUF_SIZE 200
#define VERB_BUF_SIZE 10
#define PATH_BUF_SIZE 100

#define DEFAULT_HEADERS_BUF_SIZE 10
```

```
#define HTTP_RES_STRING_BUF_SIZE 200

httpreq_t httpreq_parse_from_client(
  struct sockaddr_in *const client_address,
  socklen_t *const client_address_len,
  const int socket_fd
)
{
  httpreq_t httpreq = {UNKNOWN, NULL};

  char read_buf[READ_BUF_SIZE] = {0};

  if (recvfrom(socket_fd, read_buf, READ_BUF_SIZE, 0, (struct sockaddr
     *)client_address, client_address_len) == -1)
  return httpreq;

  char verb_buf[VERB_BUF_SIZE] = {0};
  char path_buf[PATH_BUF_SIZE] = {0};

  int read_count = sscanf(read_buf, "%s%s", verb_buf, path_buf);

  if (read_count < 2)
    return httpreq;

  if (strcmp(verb_buf, "GET") == 0)
    httpreq.verb = GET;
  else if (strcmp(verb_buf, "HEAD") == 0)
    httpreq.verb = HEAD;
  else
    return httpreq;

  if (strcmp(path_buf, "/") == 0)
    strcpy(path_buf, DEFAULT_FILE);

  size_t read_path_length = strlen(path_buf);

  char *new_path = calloc(read_path_length + 2, sizeof(char));

  if (new_path)
  {
    strcpy(new_path + 1, path_buf);
```

```
    *new_path = '.';

    httpreq.path = new_path;
  }

  return httpreq;
}

void httpreq_free(httpreq_t *const httpreq)
{
  free(httpreq->path);
  httpreq->path = NULL;
}

bool httpreq_is_null(const httpreq_t *const httpreq)
{
  return !(bool)httpreq->path;
}

httpverb_t httpreq_verb(const httpreq_t *const httpreq)
{
  return httpreq->verb;
}

const char *httpreq_path(const httpreq_t *const httpreq)
{
  return httpreq->path;
}

static httpheaders_t httpheaders_init(void)
{
  httpheaders_t httpheaders = {0, 0, NULL};

  return httpheaders;
}

httpres_t httpres_init(
  const short unsigned code,
  const char *description
)
{
```

```
  httpheaders_t httpheaders = httpheaders_init();
  httpres_t httpres = {0, NULL, httpheaders};

  size_t description_length = strlen(description);

  char *new_description = malloc(description_length);

  if (new_description)
  {
    memcpy(new_description, description, description_length);

    httpres.code = code;
    httpres.description = new_description;
  }

  return httpres;
}

static void httpheaders_free(httpheaders_t *const httpheaders)
{
  for (size_t i = 0; i < httpheaders->length; ++i)
  {
    free(httpheaders->arr[i].name);
    free(httpheaders->arr[i].value);
  }

  free(httpheaders->arr);
  httpheaders->arr = NULL;
}

void httpres_free(httpres_t *const httpres)
{
  free(httpres->description);
  httpres->description = NULL;

  httpheaders_free(&httpres->headers);
}

static int httpheaders_add_header(
  httpheaders_t *const httpheaders,
  const httpheader_t *const httpheader
```

```
)
{
  if (!httpheaders->arr)
  {
    httpheaders->arr = calloc(DEFAULT_HEADERS_BUF_SIZE, sizeof(
       httpheader_t));

    if (!httpheaders->arr)
    {
      return ERROR_HTTP_RES_ADD_HEADER;
    }

    httpheaders->buf_size = DEFAULT_HEADERS_BUF_SIZE;
  }

  if (httpheaders->length == httpheaders->buf_size)
  {
    httpheader_t *realloced_arr = realloc(httpheaders->arr, 2 *
       httpheaders->buf_size);

    if (!realloced_arr)
    {
      return ERROR_HTTP_RES_ADD_HEADER;
    }

    httpheaders->arr = realloced_arr;
    httpheaders->buf_size *= 2;
  }

  httpheaders->arr[httpheaders->length++] = *httpheader;

  return EXIT_SUCCESS;
}

int httpres_add_header(
  httpres_t *const httpres,
  const char *name,
  const char *value
)
{
  size_t name_length = strlen(name);
```

```
  size_t value_length = strlen(value);

  char *new_name = malloc(name_length);

  if (!new_name)
    return ERROR_HTTP_RES_ADD_HEADER;

  char *new_value = malloc(value_length);

  if (!new_value)
  {
    free(new_name);

    return ERROR_HTTP_RES_ADD_HEADER;
  }

  strcpy(new_name, name);
  strcpy(new_value, value);

  httpheader_t header = {new_name, new_value};

  int rc = httpheaders_add_header(&httpres->headers, &header);

  if (rc != EXIT_SUCCESS)
  {
    free(new_name);
    free(new_value);
  }

  return rc;
}

void httpres_send(
  const int socket_fd,
  struct sockaddr_in *const client_address,
  socklen_t *const client_address_len,
  const httpres_t *const httpres
)
{
  char buf[HTTP_RES_STRING_BUF_SIZE] = {0};
```

```
  sprintf(buf, "HTTP/1.1 %d %s\r\n", httpres->code, httpres->
     description);
  sendto(socket_fd, buf, strlen(buf), 0, (struct sockaddr *)
     client_address, *client_address_len);

  for (size_t i = 0; i < httpres->headers.length; ++i)
  {
    sprintf(buf, "%s: %s\r\n", httpres->headers.arr[i].name, httpres->
       headers.arr[i].value);
    sendto(socket_fd, buf, strlen(buf), 0, (struct sockaddr *)
       client_address, *client_address_len);
  }

  sendto(socket_fd, "\r\n", 2, 0, (struct sockaddr *)client_address, *
     client_address_len);
}
```

В листинге 3.3 представлена реализация модуля для обработки строк, содержащий путь к файлу.

Листинг 3.3 — Реализация модуля для обработки строк содержащий путь к файлу

```
#include "pathtools.h"

filetype_t file_type(const char *path)
{
  char *type_str = strrchr(path, '.');

  if (!type_str)
    return NONE;

  ++type_str;

  if (strcmp(type_str, FILE_TYPE_HTML) == 0)
    return HTML;

  if (strcmp(type_str, FILE_TYPE_CSS) == 0)
    return CSS;

  if (strcmp(type_str, FILE_TYPE_JS) == 0)
    return JS;

  if (strcmp(type_str, FILE_TYPE_PNG) == 0)
```

```
    return PNG;

  if (strcmp(type_str, FILE_TYPE_JPG) == 0)
    return JPG;

  if (strcmp(type_str, FILE_TYPE_JPEG) == 0)
    return JPEG;

  if (strcmp(type_str, FILE_TYPE_SWF) == 0)
    return SWF;

  if (strcmp(type_str, FILE_TYPE_GIF) == 0)
    return GIF;

  return NONE;
}

size_t file_size(const int fd)
{
  lseek(fd, 0, SEEK_SET);

  size_t size = (size_t)lseek(fd, 0, SEEK_END);

  lseek(fd, 0, SEEK_SET);

  return size;
}

bool path_is_inside(const char *path)
{
  char *path_copy = malloc((strlen(path) + 1) * sizeof(char));

  if (!path_copy)
  {
    return false;
  }

  memcpy(path_copy, path, (strlen(path) + 1) * sizeof(char));

  int dir_level = 0;
```

```
  char *dir_str = strtok(path_copy, "\\");

  while (dir_str)
  {
    if (strcmp(dir_str, ".") != 0)
      dir_level += strcmp(dir_str, "..") == 0 ? -1 : 1;

    dir_str = strtok(NULL, "\\");
  }

  return dir_level >= 0;
}
```

В листинге 3.4 представлена реализация массива файловых дескрипторов и операций над ним.

Листинг 3.4 — Реализация массива файловых дескрипторов и операций над ним

```
#include "fdsarr.h"

fdsarr_t fdsarr_alloc(const size_t buf_size)
{
  fdsarr_t fdsarr = {0, 0, NULL};

  int *arr = calloc(buf_size, sizeof(int));

  if (arr)
  {
    fdsarr.buf_size = buf_size;
    fdsarr.arr = arr;
  }

  return fdsarr;
}

void fdsarr_free(fdsarr_t *const fdsarr)
{
  free(fdsarr->arr);

  fdsarr->arr = NULL;
  fdsarr->length = fdsarr->buf_size = 0;
}
```

```
bool fdsarr_is_null(const fdsarr_t *const fdsarr)
{
  return !(bool)fdsarr->arr;
}

int fdsarr_append(fdsarr_t *const fdsarr, const int elem)
{
  if (fdsarr->length == fdsarr->buf_size)
  {
    int *realloced_arr = realloc(fdsarr->arr, 2 * fdsarr->length);

    if (!realloced_arr)
      return EXIT_FAILURE;

    fdsarr->arr = realloced_arr;
    fdsarr->buf_size *= 2;
  }

  fdsarr->arr[fdsarr->length++] = elem;

  return EXIT_SUCCESS;
}

void fdsarr_remove(fdsarr_t *const fdsarr, const int elem)
{
  size_t i = 0;

  for (; i < fdsarr->length && fdsarr->arr[i] != elem; ++i);

  if (i < fdsarr->length)
  {
    for (size_t j = i; j < fdsarr->length - 1; ++j)
    fdsarr->arr[j] = fdsarr->arr[j + 1];

    --fdsarr->length;
  }
}

void fdsarr_close(const fdsarr_t *const fdsarr)
{
  for (size_t i = 0; i < fdsarr->length; ++i)
```

```
    close(fdsarr->arr[i]);
}

size_t fdsarr_length(const fdsarr_t *const fdsarr)
{
  return fdsarr->length;
}

int fdsarr_max(const fdsarr_t *const fdsarr)
{
  if (fdsarr->length == 0)
    return -1;

  int max_fd = fdsarr->arr[0];

  for (size_t i = 1; i < fdsarr->length; ++i)
    if (max_fd < fdsarr->arr[i])
      max_fd = fdsarr->arr[i];

  return max_fd;
}
```

В листинге 3.5 представлена реализация статического веб-сервера.

Листинг 3.5 — Реализация статического веб-сервера

```
#include <fcntl.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <netinet/in.h>

#include "conf.h"
#include "logger.h"
#include "fdsarr.h"
#include "serverror.h"
#include "httptools.h"
#include "pathtools.h"

#define READ_BUF_SIZE 100
```

```
#define max(a, b) a >= b ? a : b

static int listen_fd = 0;

logger_t logger = {0, 0};

void server_shutdown(int signum)
{
  logger_info(&logger, "server shutdown");

  logger_trace(&logger, "closing listen socket");
  close(listen_fd);

  logger_trace(&logger, "closing logger");
  logger_close(&logger);

  exit(EXIT_SUCCESS);
}

static void change_sig_handlers(void)
{
  logger_trace(&logger, "setting SIGINT signal handler");
  signal(SIGINT, server_shutdown);

  logger_trace(&logger, "setting SIGTERM signal handler");
  signal(SIGTERM, server_shutdown);

  logger_trace(&logger, "setting SIGTSTP signal ignoring");
  signal(SIGTSTP, SIG_IGN);

  logger_trace(&logger, "setting SIGCHLD signal ignoring");
  signal(SIGCHLD, SIG_IGN);
}

static void init_listen_fd(void)
{
  logger_trace(&logger, "initializing server socket address struct");

  struct sockaddr_in serv_addr;
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
```

```
  serv_addr.sin_port = htons(SERVER_PORT);

  logger_trace(&logger, "creating listen cosket");
  listen_fd = socket(AF_INET, SOCK_STREAM, 0);

  if (listen_fd == -1)
  {
    logger_fatal(&logger, "error while creating listen socket");
    exit(ERROR_SOCKET_INIT);
  }

  logger_trace(&logger, "binding listen socket to server address");
  if (bind(listen_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
      == -1)
  {
    logger_fatal(&logger, "error while creating listen socket");

    logger_trace(&logger, "closing listen socket");
    close(listen_fd);

    exit(ERROR_SOCKET_BIND);
  }

  logger_trace(&logger, "marking listen socket as connection-mode");
  if (listen(listen_fd, LISTEN_MAX_CONNECTIONS) == -1)
  {
    logger_fatal(&logger, "error while marking listen socket as
       connection-mode");

    logger_trace(&logger, "closing listen socket");
    close(listen_fd);

    exit(ERROR_LISTEN_SOCKET);
  }
}

static sigset_t init_pselect_sigset(void)
{
  logger_info(&logger, "initializing pselect sigset");

  sigset_t pselect_sigmask;
```

```
  sigemptyset(&pselect_sigmask);
  sigaddset(&pselect_sigmask, SIGTERM);

  return pselect_sigmask;
}

static void send_error(
  const int fd,
  struct sockaddr_in *const client_address,
  socklen_t *const client_address_len,
  const int code,
  const char *description
)
{
  logger_info(&logger, "sending HTTP-error");

  httpres_t res = httpres_init(code, description);

  char content[100] = {0};
  sprintf(content, "<h1>%d %s</h1>", code, description);

  char content_length_str[100] = {0};
  sprintf(content_length_str, "%zu", strlen(content));

  httpres_add_header(&res, "Content-Type", "text/html");
  httpres_add_header(&res, "Content-Length", content_length_str);

  httpres_send(fd, client_address, client_address_len, &res);

  sendto(fd, content, strlen(content), 0, (struct sockaddr *)
     client_address, *client_address_len);
}

static int form_httpres_headers(
  httpres_t *const httpres,
  const char *content_type,
  const int file_fd
)
{
  logger_info(&logger, "forming http headers");
```

```
  int rc = httpres_add_header(httpres, "Content-Type", content_type);

  if (rc != EXIT_SUCCESS)
  {
    return rc;
  }

  char buf[100] = {0};
  sprintf(buf, "%zu", file_size(file_fd));

  rc = httpres_add_header(httpres, "Content-Length", buf);

  return rc;
}

static void send_file_content(
  const int client_fd,
  struct sockaddr_in *const client_address,
  socklen_t *const client_address_len,
  const int file_fd
)
{
  logger_info(&logger, "sending file content");

  char buf[READ_BUF_SIZE] = {0};

  ssize_t read_bytes = read(file_fd, buf, READ_BUF_SIZE);

  while (read_bytes > 0)
  {
    sendto(client_fd, buf, read_bytes, 0, (struct sockaddr *)
       client_address, *client_address_len);

    read_bytes = read(file_fd, buf, READ_BUF_SIZE);
  }
}

static void send_response(
  const int client_fd,
  struct sockaddr_in *const client_address,
```

```
  socklen_t *const client_address_len,
  const int file_fd,
  const filetype_t filetype,
  const httpverb_t verb
)
{
  logger_info(&logger, "sending OK http response");

  logger_trace(&logger, "send_response - initializing httpes");
  httpres_t httpres = httpres_init(HTTP_CODE_OK, "OK");

  int rc = EXIT_SUCCESS;

  logger_trace(&logger, "send_response - generating httpres headers
     according to file type");
  switch (filetype)
  {
    case NONE:
      rc = ERROR_HTTP_UKNOWN_FILE_TYPE;
      break;
    case HTML:
      rc = form_httpres_headers(&httpres, HTTP_CONTENT_TYPE_HTML,
         file_fd);
      break;
    case CSS:
      rc = form_httpres_headers(&httpres, HTTP_CONTENT_TYPE_CSS,
         file_fd);
      break;
    case JS:
      rc = form_httpres_headers(&httpres, HTTP_CONTENT_TYPE_JS, file_fd
         );
      break;
    case PNG:
      rc = form_httpres_headers(&httpres, HTTP_CONTENT_TYPE_PNG,
         file_fd);
      break;
    case JPG:
      rc = form_httpres_headers(&httpres, HTTP_CONTENT_TYPE_JPG,
         file_fd);
      break;
    case JPEG:
```

```
      rc = form_httpres_headers(&httpres, HTTP_CONTENT_TYPE_JPEG,
         file_fd);
      break;
    case SWF:
      rc = form_httpres_headers(&httpres, HTTP_CONTENT_TYPE_SWF,
         file_fd);
      break;
    case GIF:
      rc = form_httpres_headers(&httpres, HTTP_CONTENT_TYPE_GIF,
         file_fd);
      break;
  }

  if (rc == EXIT_SUCCESS)
  {
    logger_trace(&logger, "send_response - generating httpres headers
       according to file type");
    httpres_send(client_fd, client_address, client_address_len, &
       httpres);

    if (verb == GET)
    {
      send_file_content(client_fd, client_address, client_address_len,
         file_fd);
    }
  }
  else
  {
    logger_info(&logger, "internal server error while generating
       httpres headers");
    send_error(client_fd, client_address, client_address_len,
       HTTP_CODE_INTERNAL_SERVER_ERROR, "Internal server error");
  }

  httpres_free(&httpres);
}

static void perform_request(
  const int client_fd,
  struct sockaddr_in *const client_address,
  socklen_t *const client_address_len,
```

```
  const httpreq_t *const httpreq_ptr
)
{
  logger_info(&logger, "performing http request");

  logger_trace(&logger, "perform_request - checking http verb");
  httpverb_t verb = httpreq_verb(httpreq_ptr);

  if (verb != GET && verb != HEAD)
  {
    logger_info(&logger, "sending 405 Method Not Allowed");
    send_error(client_fd, client_address, client_address_len,
       HTTP_CODE_METHOD_NOT_ALLOWED, "Method not allowed");
    return;
  }

  logger_trace(&logger, "perform_request - getting http URI");
  const char *path = httpreq_path(httpreq_ptr);

  if (!path_is_inside(path))
  {
    logger_info(&logger, "sending 403 Forbidden");
    send_error(client_fd, client_address, client_address_len,
       HTTP_CODE_FORBIDDEN, "Forbidden");
    return;
  }

  logger_trace(&logger, "perform_request - opening source file for
     reading");
  int requested_fd = open(path, O_RDONLY);

  if (requested_fd == -1)
  {
    logger_info(&logger, "sending 404 Not Found");
    send_error(client_fd, client_address, client_address_len,
       HTTP_CODE_NOT_FOUND, "Not found");
    return;
  }

  logger_trace(&logger, "perform_request - getting source file type");
  filetype_t filetype = file_type(path);
```

```
  if (filetype == NONE)
  {
    logger_info(&logger, "unknown file type, sending 405 Method Not
       Allowed");
    send_error(client_fd, client_address, client_address_len,
       HTTP_CODE_METHOD_NOT_ALLOWED, "Method not allowed");
    close(requested_fd);
    return;
  }

  send_response(client_fd, client_address, client_address_len,
     requested_fd, filetype, verb);

  logger_trace(&logger, "perform_request - closing source file");
  close(requested_fd);
}

static int serve_client(const int client_fd)
{
  logger_info(&logger, "forking proccess for client service");

  int pid = -1;

  if ((pid = fork()) == -1)
  {
    logger_error(&logger, "cannot fork");

    return ERROR_FORK_PROCCESS;
  }

  if (pid == 0)
  {
    logger_info(&logger, "child process is serving clients request");

    logger_trace(&logger, "serve_client - parsing http request from
       client");

    struct sockaddr_in client_address;
    socklen_t client_address_len = sizeof(client_address);
```

```
    httpreq_t httpreq = httpreq_parse_from_client(&client_address, &
       client_address_len, client_fd);

    if (httpreq_is_null(&httpreq))
    {
      logger_error(&logger, "cannot parse http request, child process
         is beeing killed");

      exit(ERROR_HTTP_REQ_PARSE);
    }

    logger_trace(&logger, "serve_client - getting http verb");
    httpverb_t verb = httpreq_verb(&httpreq);

    perform_request(client_fd, &client_address, &client_address_len, &
       httpreq);

    logger_trace(&logger, "serve_client - free http request structure
       memory");
    httpreq_free(&httpreq);
  }

  close(client_fd);

  if (pid == 0)
    exit(EXIT_SUCCESS);

  return EXIT_SUCCESS;
}

static int connect_new_client(
  fdsarr_t *const client_fds_arr_ptr
)
{
  logger_info(&logger, "accepting new connection");

  logger_trace(&logger, "connect_new_client - accepting new client fd
     with accept");
  struct sockaddr client_addr;
  socklen_t client_len;
```

```
  int client_fd = accept(listen_fd, (struct sockaddr *)&client_addr, &
     client_len);

  if (client_fd == -1)
  {
    logger_error(&logger, "client socket accepting error");

    return ERROR_SOCKET_ACCEPT;
  }

  logger_trace(&logger, "connect_new_client - appending new client fd
     to clients fds array");
  int rc = fdsarr_append(client_fds_arr_ptr, client_fd);

  if (rc != EXIT_SUCCESS)
  {
    logger_error(&logger, "connect_new_client - cannot append new
       client fd to clients fds array");

    return ERROR_FDSARR_APPEND;
  }

  return EXIT_SUCCESS;
}

static void handle_read_sockets(
  fdsarr_t *const client_fds_arr_ptr,
  fd_set *const read_fd_set_ptr
)
{
  logger_info(&logger, "handling sockets");

  size_t i = 0;

  while (i < client_fds_arr_ptr->length)
  {
    int socket_fd = client_fds_arr_ptr->arr[i];

    if (FD_ISSET(socket_fd, read_fd_set_ptr))
    {
      FD_CLR(socket_fd, read_fd_set_ptr);
```

```
      fdsarr_remove(client_fds_arr_ptr, socket_fd);

      serve_client(socket_fd);
    }
    else
    ++i;
  }

  if (FD_ISSET(listen_fd, read_fd_set_ptr))
    connect_new_client(client_fds_arr_ptr);
}

static void update_read_fds_set(
  int *const nfds_ptr,
  fd_set *const read_fd_set_ptr,
  const fdsarr_t *const client_fds_arr_ptr
)
{
  logger_info(&logger, "updating read fds set");

  FD_ZERO(read_fd_set_ptr);

  size_t client_fds_count = fdsarr_length(client_fds_arr_ptr);

  for (size_t i = 0; i < client_fds_count; ++i)
    FD_SET(client_fds_arr_ptr->arr[i], read_fd_set_ptr);

  FD_SET(listen_fd, read_fd_set_ptr);

  *nfds_ptr = max(listen_fd, fdsarr_max(client_fds_arr_ptr));
}

int main(void)
{
  int rc = logger_init(&logger, DEFAULT_LOG_FILE_NAME,
     DEFAULT_LOG_LEVEL);

  if (rc != EXIT_SUCCESS)
    return ERROR_LOGGER_INIT;

  logger_info(&logger, "setting signals handlers");
```

```
change_sig_handlers();

logger_info(&logger, "initializing listen socket");
init_listen_fd();

logger_info(&logger, "initializing read fd set");
fd_set read_fd_set;
FD_ZERO(&read_fd_set);
FD_SET(listen_fd, &read_fd_set);

logger_info(&logger, "allocation memory for clients fds array");
fdsarr_t client_fds_arr = fdsarr_alloc(DEFAULT_FDSARR_BUF_SIZE);

if (fdsarr_is_null(&client_fds_arr))
{
  logger_fatal(&logger, "cannot allocate memory for clients fds array
     ");

  logger_info(&logger, "closing listen socket");
  close(listen_fd);

  logger_info(&logger, "closing logger");
  logger_close(&logger);

  exit(ERROR_FDSARR_ALLOC);
}

sigset_t pselect_sigmask = init_pselect_sigset();

int nfds = listen_fd;

logger_info(&logger, "running server");

while (1)
{
  logger_info(&logger, "searching for fds with pselect");
  rc = pselect(nfds + 1, &read_fd_set, NULL, NULL, NULL, &
     pselect_sigmask);

  if (rc == -1)
  {
```

```
      logger_info(&logger, "closing listen socket");
      close(listen_fd);

      logger_info(&logger, "closing read fds from fdsarr");
      fdsarr_close(&client_fds_arr);

      logger_info(&logger, "free fdsarr allocated memory");
      fdsarr_free(&client_fds_arr);

      logger_info(&logger, "closing logger");
      logger_close(&logger);

      exit(ERROR_PSELECT_READ);
    }

    handle_read_sockets(&client_fds_arr, &read_fd_set);

    update_read_fds_set(&nfds, &read_fd_set, &client_fds_arr);
  }

  return EXIT_SUCCESS;
}
```

Вывод

Были выбраны средства реализации статического веб-сервера. Был написан код программного обеспечения для отдачи контента с диска по HTTP-запросу.

4 Исследовательская часть

Целью исследования является сравнение результатов нагрузочного тестирования разработанного программного обеспечения и сервера, развернутого на базе nginx [12].

Для проведения исследования была использована электронная вычислительная машина, обладающая следующими характеристиками:

— операционная система Manjaro Linux x86_64 [13];

— процессор Intel i7-10510U 4.900 ГГц [14];

— оперативная память DDR4, 2400 МГц, 8 ГБ [15].

Нагрузочное тестирование проводилось с помощью утилиты Apache Benchmark [16].

В таблицах 4.1 и 4.2 представлены результаты нагрузочного тестирования сервера, развернутого на базе nginx, и разработанного программного обеспечения соответственно.

Таблица 4.1 — Результаты нагрузочного тестирования сервера, развернутого на базе nginx

Общее число запросов, шт.	Число запросов, отправляемых за раз, шт.	Общее время тестирования, с	Среднее число запросов в секунду, шт.	Среднее время выполнения одного запроса, мс
100	1	0.011	9175.15	0.109
100	5	0.005	19669.55	0.051
100	10	0.006	17452.01	0.057
500	1	0.040	12421.74	0.081
500	5	0.029	17480.07	0.057
500	10	0.021	24112.65	0.041
1000	1	0.075	13345.79	0.075
1000	5	0.041	24262.42	0.041
1000	10	0.036	27480.83	0.036
10000	1	0.664	15059.79	0.066
10000	5	0.315	31741.50	0.032
10000	10	0.336	29731.91	0.034
25000	1	1.520	16450.67	0.061
25000	5	0.801	31202.26	0.032
25000	10	0.795	31443.38	0.032

Таблица 4.2 — Результаты нагрузочного тестирования разработанного сервера

Общее число запросов, шт.	Число запросов, отправляемых за раз, шт.	Общее время тестирования, с	Среднее число запросов в секунду, шт.	Среднее время выполнения одного запроса, мс
100	1	0.041	2419.96	0.413
100	5	0.017	5776.67	0.173
100	10	0.016	6239.47	0.160
500	1	0.252	1984.82	0.394
500	5	0.065	7640.47	0.131
500	10	0.074	6800.50	0.147
1000	1	0.369	2711.27	0.369
1000	5	0.143	7001.33	0.143
1000	10	0.140	7158.81	0.140
10000	1	3.190	3134.62	0.319
10000	5	1.362	7344.42	0.136
10000	10	1.354	7385.13	0.135
25000	1	7.759	3221.88	0.310
25000	5	3.371	7415.68	0.135
25000	10	3.334	7498.62	0.133

На рисунках 4.1 – 4.3 представлены графики зависимости среднего времени выполнения запроса от общего количества запросов.

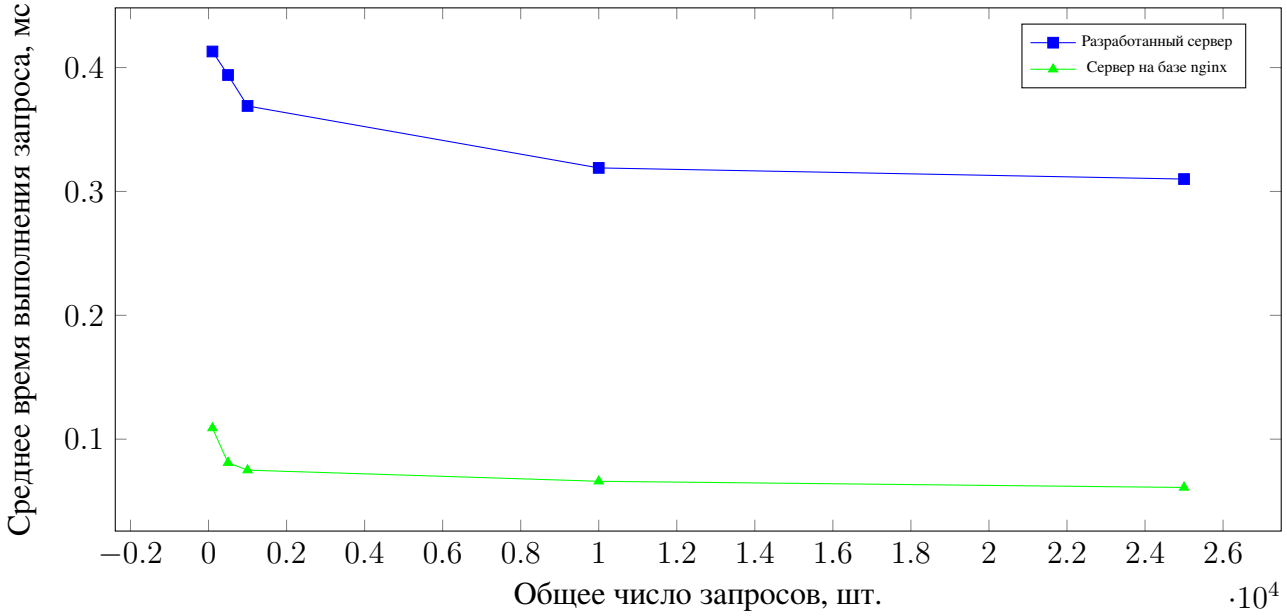

Рисунок 4.1 — Графики зависимости среднего времени выполнения запроса от общего количества запросов (число запросов за раз — 1 шт.)

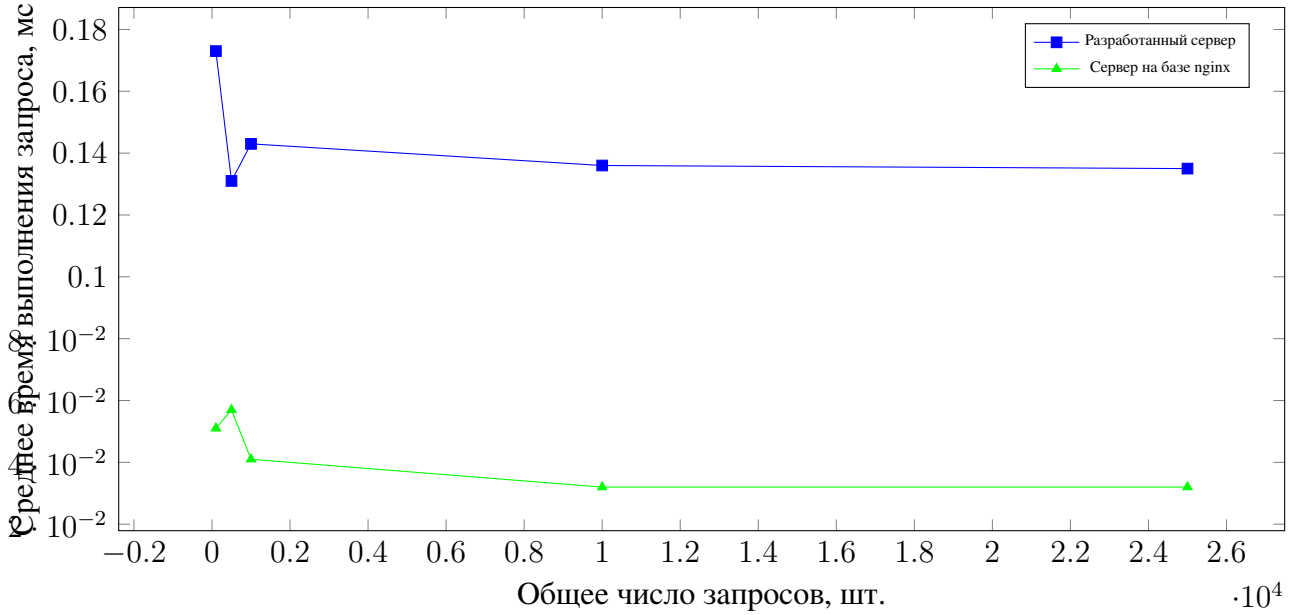

Рисунок 4.2 — Графики зависимости среднего времени выполнения запроса от общего количества запросов (число запросов за раз — 5 шт.)

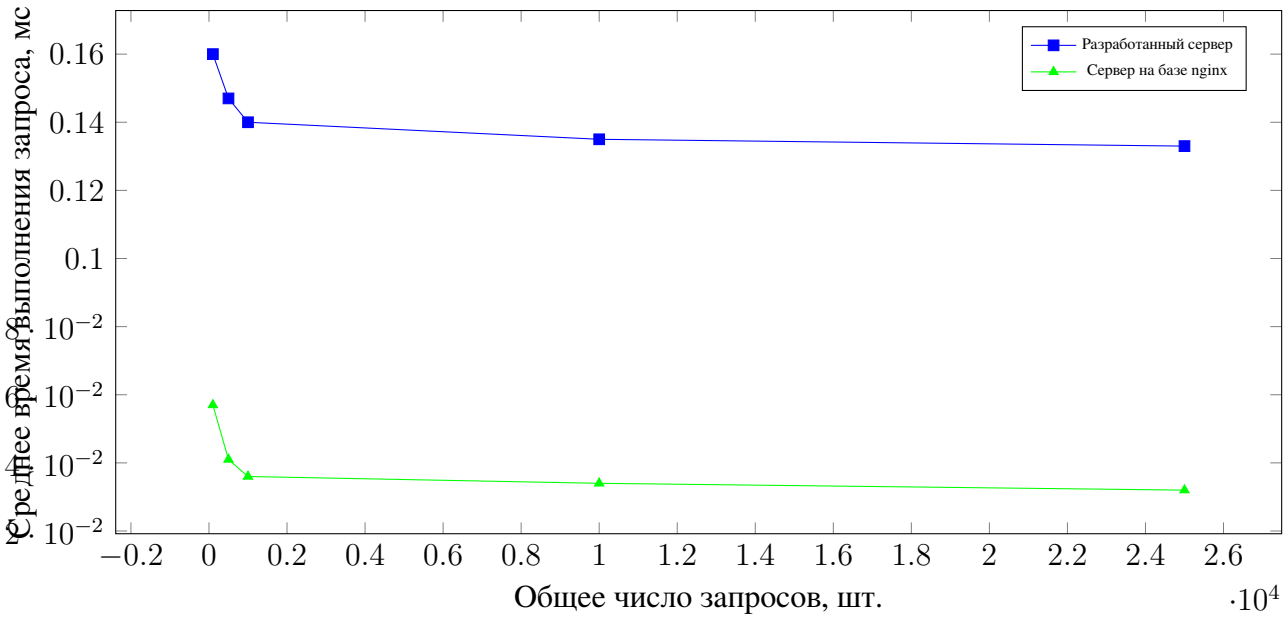

Рисунок 4.3 — Графики зависимости среднего времени выполнения запроса от общего количества запросов (число запросов за раз — 10 шт.)

4.1 Вывод

Было проведено сравнение результатов нагрузочного тестирования разработанного программного обеспечения и сервера, развернутого на базе nginx. Согласно полученным данным, сервер на базе nginx выполняет запросы в среднем в 2.5 раза быстрее, чем разработанное программное обеспечение.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был реализован классической статический веб-сервер для отдачи контента с диска.

Была изучена предметная область, связанная со статическим веб-сервером. Был проведен анализ протокола HTTP. Была осуществлена формализация бизнес-правил разрабатываемого программного обеспечения.

Были сформулированы требования к разрабатываемому статическому веб-серверу. Были проанализированы сокеты как средство взаимодействия между процессами. Были разработаны схемы алгоритмов работы статического веб-сервера и обработки HTTP-запросов.

Были выбраны средства реализации статического веб-сервера. Был написан код программного обеспечения для отдачи контента с диска по HTTP-запросу.

Было проведено сравнение результатов нагрузочного тестирования разработанного программного обеспечения и сервера, развернутого на базе nginx. Согласно полученным данным, сервер на базе nginx выполняет запросы в среднем в 2.5 раза быстрее, чем разработанное программное обеспечение.

Были выполнены следующие задачи:

— анализ предметной области, связанной со статическим веб-сервером;
— предъявление требований к разрабатываемому программному обеспечению;
— проектирование архитектуры статического веб-сервера для отдачи контента с диска;
— реализация статического веб-сервера для отдачи контента с диска;
— исследование характеристик реализованного сервера.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Что такое веб-сервер — Изучение веб-разработки | MDN [Электронный ресурс] — Режим доступа: https://developer.mozilla.org/ru/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server (дата обращения 26.01.2023)

2. RFC 2616 — Hypertext Transfer Protocol — HTTP/1.1 [Электронный ресурс] — Режим доступа: https://datatracker.ietf.org/doc/html/rfc2616 (дата обращения 26.01.2023)

3. Сокеты — Сетевое программирование [Электронный ресурс] — Режим доступа: https://datatracker.ietf.org/doc/html/rfc2616 (дата обращения 26.01.2023)

4. socket(2) — Linux manual page [Электронный ресурс] — Режим доступа: https://man7.org/linux/man-pages/man2/socket.2.html (дата обращения 10.02.2023)

5. bind(2) — Linux manual page [Электронный ресурс] — Режим доступа: https://man7.org/linux/man-pages/man2/bind.2.html (дата обращения 10.02.2023)

6. listen(2) — Linux manual page [Электронный ресурс] — Режим доступа: https://man7.org/linux/man-pages/man2/listen.2.html (дата обращения 10.02.2023)

7. accept(2) — Linux manual page [Электронный ресурс] — Режим доступа: https://man7.org/linux/man-pages/man2/accept.2.html (дата обращения 10.02.2023)

8. recvfrom(3p) — Linux manual page [Электронный ресурс] — Режим доступа: https://man7.org/linux/man-pages/man3/recvfrom.3p.html (дата обращения 10.02.2023)

9. sendto(3p) — Linux manual page [Электронный ресурс] — Режим доступа: https://man7.org/linux/man-pages/man3/sendto.3p.html (дата обращения 10.02.2023)

10. fork(2) — Linux manual page [Электронный ресурс] — Режим доступа: https://man7.org/linux/man-pages/man2/fork.2.html (дата обращения 10.02.2023)

11. pselect(3p) — Linux manual page [Электронный ресурс] — Режим доступа: https://man7.org/linux/man-pages/man3/pselect.3p.html (дата обращения 10.02.2023)

12. nginx [Электронный ресурс] — Режим доступа: https://nginx.org/ (дата обращения 10.02.2023)

13. Manjaro [Электронный ресурс] — Режим доступа: https://manjaro.org/ (дата обращения 10.02.2023)

14. Intel Core i7-10510U Processor [Электронный ресурс] — Режим доступа: https://www.intel.com/content/www/us/en/products/sku/196449/intel-core-i710510u-processor-8m-cache-up-to-4-90-ghz/downloads.html#!=www.intel.com-# (дата обращения 10.02.2023)

15. HP ProBook 430 G7 Notebook PC Specifications [Электронный ресурс] — Режим доступа: https://support.hp.com/my-en/document/c06469987 (дата обращения 10.02.2023)

16. ab — Apache HTTP server benchmarking tool [Электронный ресурс] — Режим доступа: https://httpd.apache.org/docs/trunk/programs/ab.html (дата обращения 10.02.2023)