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 группы **ИУ7-73Б** _____ **Неумоин Д.Ю.**

(Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ _____

(Подпись, дата) (И.О. Фамилия)

***202_** г.*

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Термин «веб-сервер» может относиться как к аппаратному, так и к программному обеспечению.

С точки зрения аппаратного обеспечения веб-сервер – это компьютер, на котором хранятся программное обеспечение веб-сервера и файлы компонентов веб-сайта (например, HTML-документы, изображения, стили CSS и файлы JavaScript). Веб-сервер подключается к Интернету и поддерживает физический обмен данными с другими устройствами, подключенными к Интернету.

С точки зрения программного обеспечения веб-сервер включает в себя различные компоненты, которые контролируют доступ веб-пользователей к размещённым файлам. Одним из таких компонентов является HTTP-сервер.

HTTP-сервер – это программное обеспечение, которое работает с URL-адресами по протоколу HTTP, доставляя содержимое веб-сайтов на устройства конечных пользователей. Доступ к HTTP-серверу можно получить через доменные имена веб-сайтов, которые он хранит.

Статический сервер – это сервер, который предоставляет клиентам исключительно статические файлы. Это могут быть файлы HTML, CSS, JavaScript, изображения, видео, аудио и любые другие файлы, которые клиенты скачивают и используют в исходном виде, без какой-либо обработки на стороне сервера.

Статические серверы существенно отличаются от динамических серверов, которые могут выполнять серверный код, чтобы формировать вебстраницы «на лету» или предоставлять другие функции, такие как обработка форм, взаимодействие с базами данных и выполнение других операций.

Целью данной работы является разработка статического сервера. Для достижения поставленной цели необходимо решить следующие задачи.

1) Провести анализ предметной области и формализовать задачу.
2) Спроектировать структуру программного обеспечения.
3) Реализовать программное обеспечение, которое будет обслуживать кон-

тент, хранящийся во вторичной памяти.

4) Провести нагрузочное тестирование и сравнение с распространёнными аналогами.

1 Аналитическая часть

В данном разделе будут описаны основные теоретические аспекты, необходимые для решения поставленной задачи

1.1 Протокол HTTP

HTTP (англ. Hypertext Transfer Protocol, гипертекстовый транспортный протокол) – протокол, определяющий набор соглашений по передаче гипертекста (т.е. связанных веб-документов) между двумя компьютерами. HTTP является текстовым протоколом без сохранения состояния. **??**

Данный протокол задаёт следующие правила взаимодействия клиента и сервера.

- HTTP-запросы производятся исключительно от клиентов к серверу, который способен только отвечать на запросы.
- При запросе файла по HTTP клиент должен сформировать файловый URL (Uniform Resource Locator, единообразный указатель местонахождения ресурса).
- Веб-сервер должен ответить на каждый HTTP-запрос, даже в случае возникновения ошибки.

HTTP-запросы и ответы имеют схожую структуру и состоят из следующих элементов.

- Стартовая строка, содержащая метод HTTP-запроса, цель запроса, версия протокола.
- Необязательный набор заголовков HTTP, определяющий запрос или описывающий тело сообщения.
- Необязательное тело сообщения, содержащее данные, связанные с запросом (например, содержимое HTML-формы), или документ, связанный с ответом. Наличие тела и его размер задаются начальной строкой и заголовками HTTP.

Начальная строка и HTTP-заголовки HTTP-сообщения вместе называются заголовком, а его полезная нагрузка – телом. На рисунке 1 представлен пример HTTP запроса и ответа версии 1.x.

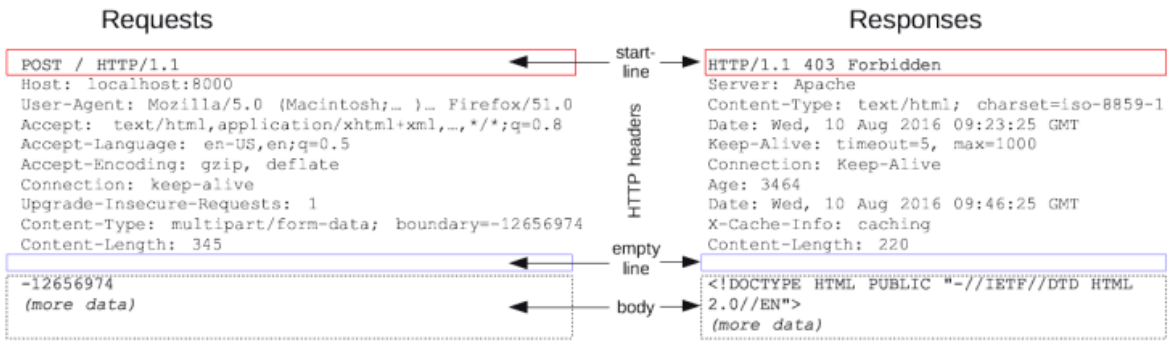

Рисунок 1 – Пример запроса и ответа HTTP.

1.2 Веб-серверы

Веб-серверы, как правило, работают по протоколам HTTP/HTTPS и предоставляют клиентам доступ к файлам, таким как веб-страницы. Чтобы загрузить веб-страницу, браузер отправляет запрос к веб-серверу, который выполняет поиск запрашиваемого файла в своём хранилище. Найдя файл, сервер считывает его, обрабатывает при необходимости и отсылает в браузер.

Такие серверы могут работать со статическим и динамическим контентом: статический отдаётся клиенту в исходном виде без изменений, а динамический является результатом обработки данных на стороне сервера-приложения. Со статическим контентом проще работать, в то время, как динамический контент обеспечивает большую гибкость.

Веб-серверы могут являтся связующим звеном между клиентами, например браузерами, и серверами-приложенями, проксируя трафик.

1.3 Сокеты

Сокет — это абстракция конечной точки соединения, которая используется для обеспечения обмена данными между устройствами сети. Сокеты яв-

ляются ключевым компонентом для установки и управления сетевыми соединениями.

Сокеты предоставляют интерфейс для создания конечных точек соединения в сети с использованием протоколов передачи данных, таких как TCP или UDP. При разработке веб-серверов сокеты используются для «прослушивания» входящих соединений от клиентов и передачи данных между сервером и клиентами.

Процесс создания сервера с использованием сокетов включает в себя следующие шаги.

1. Создание сокета, который будет слушать входящие соединения.
2. Привязка сокета к адресу и порту: после создания сокета необходимо привязать его к сетевому адресу и порту, на котором будет прослушиваться входящий трафик.
3. Установка сервера в состояние прослушивания входящих соединений.
4. Принятие входящего соединения.
5. Обработка запросов и передача данных. После установки соединения с клиентом сервер может принимать запросы от клиента с помощью функций чтения и записи данных через сокет.

Использование сокетов в разработке серверов позволяет эффективно управлять сетевыми соединениями и обеспечивать передачу статического контента клиентам по сети.

1.4 Мультиплексирование ввода-вывода

Мультиплексирование ввода-вывода является методом обработки нескольких операций ввода-вывода в одном потоке выполнения программы, что позволяет повысить эффективность управления множеством соединений в сетевых приложениях или приложениях с асинхронным вводом-выводом. Это позволяет уменьшить задержки и повысить производительность.

Мультиплексоры позволяют приложению ожидать ввода или вывода данных из нескольких источников, таких как сокеты, файлы или сетевые устрой-

ства, используя один системный вызов вместо создания или управления каждым потоком или процессом отдельно.

Использование мультиплексирования ввода-вывода требует более сложной логики обработки событий, чем простое параллельное программирование, но оно может обеспечить более эффективное использование системных ресурсов и более высокую производительность.

Существуют следующие механизмы мультиплексирования сетевых соединений.

select

Это системный вызов, используемый в различных операционных системах для ожидания событий на нескольких файловых дескрипторах, таких как сокеты. Он позволяет одному потоку контролировать несколько соединений одновременно. Данный мультиплексор имеет ограничение на максимальное количество отслеживаемых файловых дескрипторов — 1024.

pselect

pselect схож по принципу работы с select и имеет те же ограничения, однако реализует более продвинутую обработку сигналов. В отличие от select, pselect позволяет процессу заблокироваться на ожидании событий ввода-вывода и одновременно игнорировать определенные сигналы или обрабатывать их в специфическим способом.

epoll

Системный вызов, предоставляемый в ядре Linux, обеспечивает более гибкие варианты работы с событиями ввода-вывода за счёт реализации модели уведомлений. В отличие от select и pselect, epoll предоставляет более эффективный способ мониторинга множества файловых дескрипторов на предмет готовности к вводу или выводу и не имеет ограничений на обработку файловых дескрипторов

epoll является наиболее современным и эффективным механизмом в сравнении с select и pselect и часто используется в высоконагруженных сетевых при-

ложениях на ОС Linux.

1.5 Распараллеливание обработки входящих запросов

Для ускорения обработки запросов веб-серверы реализуют их параллельную обработку в многопроцессорной среде. Существует множество подходов к параллелизации работы веб-серверов, в данной работе будут рассмтрены: пул потоков (thread pool), разветвление (prefork) и (thread per request).

Thread Pool представляет собой набор потоков, готовых к выполнению задач. Когда в систему поступает новая задача, она помещается в очередь, и один из доступных потоков в пуле забирает эту задачу на выполнение. После завершения задачи поток возвращается обратно в пул и становится доступным для выполнения новых задач. Это способствует уменьшению накладных расходов на создание и уничтожение потоков, а также позволяет управлять ресурсами более эффективно и обеспечивает параллельное выполнение задач.

Prefork – это модель параллелизации, при которой веб-сервер создаёт отдельные процессы для обработки запросов. Такой подход позволяет изолировать обработку каждого запроса и повышает надежность веб-сервера, так как сбои в одном процессе не влияют на остальные. Однако создание процессов требует больше дополнительных ресурсов, чем использование пула потоков.

Thread per request – эта модель обрабатывает каждый запрос от клиента в отдельном потоке управления. Это менее эффективно для запросов небольшой продолжительности из-за затрат на создание нового потока для каждого запроса. Он также может потреблять большое количество ресурсов ОС, если множество клиентов одновременно отправляют запросы.

Выбор между подходами зависит от конкретных требований веб-сервера, предполагаемой нагрузки и характера обрабатываемых запросов.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритма обработки клиентских запросов.

2.1 Сервер

На рисунке 2 представлена схема алгоритма работы сервера, обслуживающего клиентские запросы с использованием сокетов.

В целях повышения масштабируемости приложения алгоритм работы сервера не зависит от подхода к параллельной обработке запросов. Добавить параллелизацию можно с использованием паттерна проектирования «инъекция зависимостей» (dependency injection).

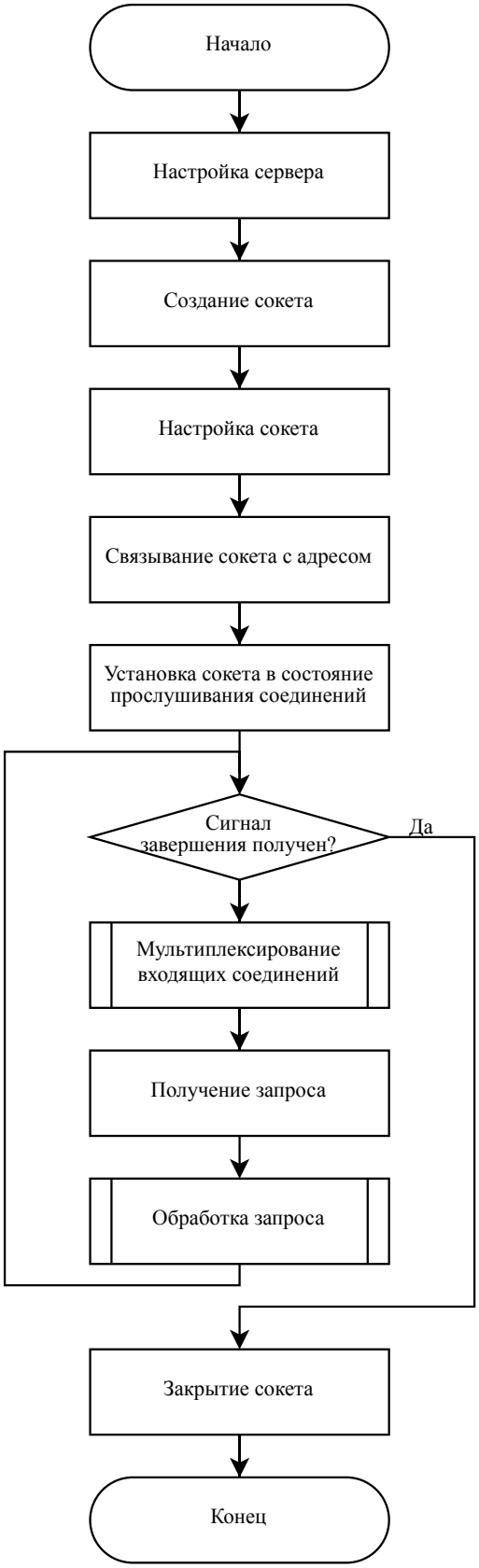

Рисунок 2 – Алгоритм работы сервера

2.2 Пул потоков

На рисунке 3 представлена схема алгоритма работы пула потоков.

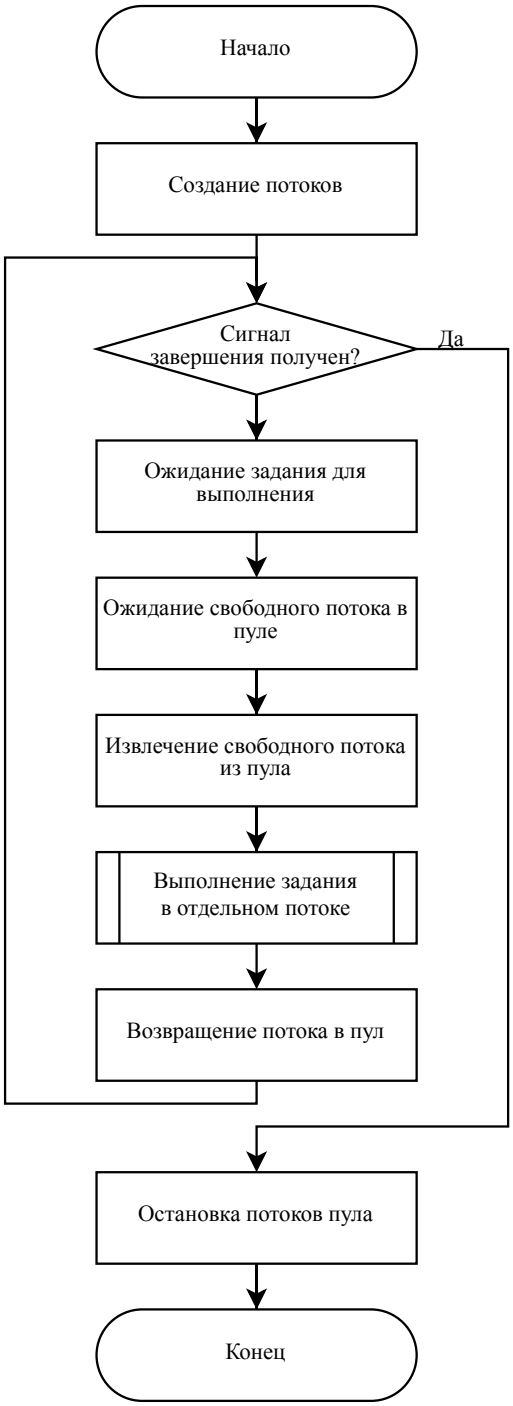

Рисунок 3 – Алгоритм работы пула потоков

3 Технологическая часть

В данном разделе будут описаны детали реализации и развёртывания ПО, а также приведены листинги кода.

3.1 Средства реализации

Для реализации статического сервера был выбран язык C в соответствии с заданием на курсовую работу. Для мультиплексирования клиентских соединений был выбран мультиплексор epoll, для параллелизации обработки запросов – пул потоков. Для контроля качества кода использовался отладчик использования памяти valgrind [2], для отладки работы с потоками использовалась утилита helgrind[3], входящая в valgrind.

3.2 Детали реализации

3.2.1 Сервер

В листингах 1 и 2 представлены функции, реализующие обработку клиентских запросов с использованием мультиплексора epoll.

Листинг 1: Функция создания серверного сокета.

```
int epoll_socket_create(char *host, int port) {
    int socket_fd;
    struct sockaddr_in server_addr;

    if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
     ↪  {
        LOG_ERROR("socket error");
        return -1;
    }

    int opt = IP_PMTUDISC_WANT;
    if (setsockopt(socket_fd, SOL_SOCKET,
                   SO_REUSEADDR | SO_REUSEPORT, &opt,
                    ↪  sizeof(opt))) {
```

```
13              LOG_ERROR("setsockopt error");
14              return -1;
15          }
16
17
18          server_addr.sin_family = AF_INET;
19          server_addr.sin_addr.s_addr = inet_addr(host);
20          server_addr.sin_port = htons(port);
21
22          if (bind(socket_fd, (struct sockaddr *)&server_addr,
23                   sizeof(server_addr)) < 0) {
24              perror("bind error");
25              return -1;
26          }
27
28          if (listen(socket_fd, CONN_REQ_QUEUE) == -1) {
29              perror("listen error");
30              return -1;
31          }
32
33          setnonblocking(socket_fd);
34
35          return socket_fd;
36      }
```

Листинг 2: Функция, содержащая основной цикл сервера с обработкой соединений.

```
38  int epoll_loop(server_t *server) {
39      struct sockaddr_in client_addr;
40      socklen_t client_len = sizeof(client_addr);
41      int client_socket, epoll_fd, nfds;
42      struct epoll_event event, events[MAX_EVENTS];
43
44      epoll_fd = epoll_create1(0);
45      if (epoll_fd == -1) {
46          perror("epoll_create1");
47          return EXIT_FAILURE;
48      }
49
50      setnonblocking(server->socket_fd);
51
52      event.data.fd = server->socket_fd;
53      event.events = EPOLLIN;
54      if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD,
        ↪  server->socket_fd, &event) ==
55          -1) {
56          perror("epoll_ctl: server_socket");
57          return EXIT_FAILURE;
58      }
59
60      server->is_running = 1;
61      while (server->is_running) {
62          nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
63          if (nfds == -1) {
64              perror("epoll_wait");
65              close(server->socket_fd);
66              break;
```

```
67              }
68
69              for (int i = 0; i < nfds; i++) {
70                  if (events[i].data.fd == server->socket_fd) {
71                      client_socket =
72                          accept(server->socket_fd,
73                                 (struct sockaddr *)&client_addr,
                                   ↪  &client_len);
74                      if (client_socket == -1) {
75                          perror("accept");
76                          continue;
77                      }
78
79                      event.data.fd = client_socket;
80                      event.events = EPOLLIN | EPOLLET;
81
82                      if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD,
                         ↪  client_socket, &event) ==
83                          -1) {
84                          perror("epoll_ctl: client_socket\n");
85                          return EXIT_FAILURE;
86                      }
87                  } else {
88                      conn_data_t *arg =
                         ↪  malloc(sizeof(conn_data_t));
89                      arg->clientfd = events[i].data.fd;
90                      arg->static_dir = server->static_dir;
91                      if (threadpool_add(server->tpool,
                         ↪  &conn_handler, (void *)arg, 0) < 0) {
92                          perror("thpool_add_work err\n");
93                          return EXIT_FAILURE;
94                      }
95                  }
96              }
97          }
```

```
98  }
```

3.2.2 Пул потоков

В листингах 3 - 5 представлены функции, реализующие пул потоков.

Листинг 3: Функция создания пула потоков.

```
1   threadpool_t *threadpool_create(int thread_count, int
    ↪  queue_size, int flags)
2   {
3       threadpool_t *pool;
4       int i;
5       (void) flags;
6
7       if(thread_count <= 0 || thread_count > MAX_THREADS ||
        ↪  queue_size <= 0 || queue_size > MAX_QUEUE) {
8           return NULL;
9       }
10
11      if((pool = (threadpool_t *)malloc(sizeof(threadpool_t)))
        ↪  == NULL) {
12          goto err;
13      }
14
15      /* Initialize */
16      pool->thread_count = 0;
17      pool->queue_size = queue_size;
18      pool->head = pool->tail = pool->count = 0;
19      pool->shutdown = pool->started = 0;
20
21      /* Allocate thread and task queue */
22      pool->threads = (pthread_t *)malloc(sizeof(pthread_t) *
        ↪  thread_count);
```

```
23      pool->queue = (threadpool_task_t *)malloc
24          (sizeof(threadpool_task_t) * queue_size);
25
26      /* Initialize mutex and conditional variable first */
27      if((pthread_mutex_init(&(pool->lock), NULL) != 0) ||
28         (pthread_cond_init(&(pool->notify), NULL) != 0) ||
29         (pool->threads == NULL) ||
30         (pool->queue == NULL)) {
31          goto err;
32      }
33
34      /* Start worker threads */
35      for(i = 0; i < thread_count; i++) {
36          if(pthread_create(&(pool->threads[i]), NULL,
37                            threadpool_thread, (void*)pool) !=
                               ↪  0) {
38              threadpool_destroy(pool, 0);
39              return NULL;
40          }
41          pool->thread_count++;
42          pool->started++;
43      }
44
45      return pool;
46
47   err:
48      if(pool) {
49          threadpool_free(pool);
50      }
51      return NULL;
52  }
```

Листинг 4: Функция добавления задачи в очередь на выполнение.

```
54  int threadpool_add(threadpool_t *pool, void (*function)(void
     ↪  *),
55                     void *argument, int flags)
56  {
57      int err = 0;
58      int next;
59      (void) flags;
60
61      if(pool == NULL || function == NULL) {
62          return threadpool_invalid;
63      }
64
65      if(pthread_mutex_lock(&(pool->lock)) != 0) {
66          return threadpool_lock_failure;
67      }
68
69      next = (pool->tail + 1) % pool->queue_size;
70
71      do {
72          /* Are we full ? */
73          if(pool->count == pool->queue_size) {
74              err = threadpool_queue_full;
75              break;
76          }
77
78          /* Are we shutting down ? */
79          if(pool->shutdown) {
80              err = threadpool_shutdown;
81              break;
82          }
83
84          /* Add task to queue */
85          pool->queue[pool->tail].function = function;
```

```
86              pool->queue[pool->tail].argument = argument;
87              pool->tail = next;
88              pool->count += 1;
89
90              /* pthread_cond_broadcast */
91              if(pthread_cond_signal(&(pool->notify)) != 0) {
92                  err = threadpool_lock_failure;
93                  break;
94              }
95          } while(0);
96
97          if(pthread_mutex_unlock(&pool->lock) != 0) {
98              err = threadpool_lock_failure;
99          }
100
101         return err;
102     }
```

Листинг 5: Функция обработки очереди задач.

```
149 static void *threadpool_thread(void *threadpool)
150 {
151     threadpool_t *pool = (threadpool_t *)threadpool;
152     threadpool_task_t task;
153
154     for(;;) {
155         /* Lock must be taken to wait on conditional
            ↪  variable */
156         pthread_mutex_lock(&(pool->lock));
157
158         /* Wait on condition variable, check for spurious
            ↪  wakeups.
```

```
159             When returning from pthread_cond_wait(), we own
                ↪  the lock. */
160             while((pool->count == 0) && (!pool->shutdown)) {
161                 pthread_cond_wait(&(pool->notify),
                    ↪  &(pool->lock));
162             }
163
164             if((pool->shutdown == immediate_shutdown) ||
165                ((pool->shutdown == graceful_shutdown) &&
166                 (pool->count == 0))) {
167                 break;
168             }
169
170             /* Grab our task */
171             task.function = pool->queue[pool->head].function;
172             task.argument = pool->queue[pool->head].argument;
173             pool->head = (pool->head + 1) % pool->queue_size;
174             pool->count -= 1;
175
176             /* Unlock */
177             pthread_mutex_unlock(&(pool->lock));
178
179             /* Get to work */
180             (*(task.function))(task.argument);
181         }
182
183         pool->started--;
184
185         pthread_mutex_unlock(&(pool->lock));
186         pthread_exit(NULL);
187         return(NULL);
188     }
```

3.3 Поддерживаемые запросы

Разработанный веб-сервер обрабатывает запросы GET и HEAD. В первом случае клиент получает в теле ответа запрошенный файл, во втором — только заголовки ответа Content-Type и Content-Length. Ниже перечислены поддерживаемые статусы ответов сервера.

— 200 — успешное завершение обработки запроса.

— 403 — доступ к запрошенному файлу запрещён.

— 404 — запрашиваемый файл не найден.

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

— 500 — внутренняя ошибка сервера.

Разработанный сервер поддерживает следующие форматы файлов (типы контента, Content-Type):

— html (text/html);

— css (text/css);

— js (text/javascript);

— png (image/png);

— jpg (image/jpg);

— jpeg (image/jpe);

— gif (image/gif);

— svg (image/svg);

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

3.4 Сборка и развертывание

Запуск разработанного приложения осуществлялся с помощью системы контейнеризации Docker. Файл для сборки образа приложения представлен в листинге 6, файл docker-compose для развертывания в листингке 7.

Листинг 6: Dockerfile образа приложения.

```
FROM gcc:latest

COPY . .

RUN gcc -o ./app -I./server/include ./server/src/*.c
 ↪  -pthread -Wpedantic
EXPOSE 8080

CMD ["./app"]
```

Листинг 7: docker-compose файл для развертывания приложения.

```
version: "3.5"
services:
  app:
    build: .
    container_name: app
    restart: always
    cpus: 4
    ulimits:
      nofile:
        soft: 65535
        hard: 65535
    ports:
      - "8083:8080"
```

4 Исследовательская часть

В данном разделе будет проведено нагрузочное тестирование разработанного сервера и сравнение его показателей производительности с NGINX.

4.1 Описание измерений

В качестве эталона для сравнения показателей производительности статического сервера был выбран NGINX [7], являющийся HTTP-сервером общего назначения. Конфигурация NGINX приведена в листинге 8, docker-compose файл для развертывания nginx в листинге 9

Листинг 8: Конфигурация NGINX.

```
server {
    listen 80;
    location /tmp/static {
        alias /tmp/static;
        include /etc/nginx/mime.types;
        autoindex on;
        autoindex_exact_size off;
        autoindex_localtime on;
    }
}
```

Листинг 9: docker-compose файл для развертывания nginx.

```
version: "3.5"
services:
  nginx:
    image: nginx
    container_name: nginx
    restart: always
    cpus: 4
    ulimits:
      nofile:
        soft: 65535
        hard: 65535
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d
```

Ниже приведены технические характеристики устройства, на котором проводилось тестирование.

- Операционная система: macos 15.
- Объём оперативной памяти: 32 Гб.
- Процессор: M1.

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования. Для замеров метрик производительности сравниваемых веб-серверов использовалась утилита Apache Benchmark (ab)[4].

4.2 Результаты измерений

Результаты нагрузочного тестирования разработанного сервера и NGINX представлены в таблицах 1-3. Для оценки производительности измерялось количество обработанных запросов в секунду (Requests Per Second, RPS).

Таблица 1 – RPS при обработке 10000 запросов на файл размером 10 Кб

Число клиентов	Разработанный сервер	NGINX
10	4927.22	3310.99
100	3386.79	3785.10
300	900.58	1001.57

Таблица 2 – RPS при обработке 1000 запросов на файл размером 1 Мб

Число клиентов	Разработанный сервер	NGINX
10	262.78	238.59
100	232.62	278.33
300	29.93	147.92

Таблица 3 – RPS при обработке 10 запросов на файл размером 104 Мб

Число клиентов	Разработанный сервер	NGINX
1	1.65	2.99
5	3.01	3.03
10	3.00	3.04

4.3 Выводы

При обработке относительно лёгких запросов RPS сравниваемых серверов не отличался более чем на 10-15%, в некоторых случаях, разработанный сервер даже превосходит NGINX, однако при увеличении числа конкурентных

запросов производительность разработанный сервер проигрывает по эффективности серверу NGINX. При увеличении размера запрашиваемых файлов производительность NGINX оказалась выше на 10-15%, при условии 100 конкуретных запросов. При работе с относительно большими файлами(100мб) оба сервера показали примерно одинаковую производительность и пропускную способность.

ЗАКЛЮЧЕНИЕ

В рамках курсовой работы был разработан статический сервер, который на большинстве сценариев использования оказался сравнимым по производительности с NGINX.

В ходе выполнения данной работы были решены следующие задачи.

1. Проведён анализ предметной области и формализована задача.
2. Спроектирована структура программного обеспечения.
3. Реализовано программное обеспечение, которое обслуживает контент, хранящийся во вторичной памяти.
4. Проведено нагрузочное тестирование и сравнение с распространёнными аналогами.

Таким образом, все поставленные задачи были выполнены, поставленная цель достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. What is a web server? [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server, свободный – (15.10.2023)

2. Valgrind [Электронный ресурс]. – Режим доступа: https://valgrind.org, свободный – (15.10.2023)

3. Helgrind: a thread error detector [Электронный ресурс]. – Режим доступа: https://valgrind.org/docs/manual/hg-manual.html, свободный – (15.10.2023)

4. ab - Apache HTTP server benchmarking tool [Электронный ресурс]. – Режим доступа: https://httpd.apache.org/docs/2.4/programs/ab.html, свободный – (15.10.2023)

ПРИЛОЖЕНИЕ А