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:

«Сервер для отдачи статического содержимого с диска (вариант №7)»

Студент ИУ7-73Б (Группа) _____ (Подпись, дата) А. А. Сёмина (И. О. Фамилия)

Руководитель курсовой работы _____ (Подпись, дата) М. Н. Клочков (И. О. Фамилия)

2024 г.

РЕФЕРАТ

Расчетно-пояснительная записка 24 с., 5 рис., 1 табл., 11 источн., 1 прил.

В данной работе представлена реализация сервера для отдачи статического содержимого с диска для 7 варианта: с использованием архитектуры разрабатываемого сервера thread pool + epoll().

Ключевые слова: epoll, thread pool, TCP, HTTP, сокет, статический сервер, мультиплексирование, Nginx.

В ходе данной работы был изучен функционал и состав статического сервера, описан принцип его работы, а также выполнена его реализация. Архитектура сервера выполнена с использованием мультиплексирования. В рамках исследования реалихованного статического сервера по варианту было проведено нагрузочное тестирование производительности. Выполнен сравнительный анализ разработанного решения и сервера Nginx, основанный на результатах тестирования.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

С ростом числа интернет-сервисов и увеличением объёмов данных, передаваемых через сеть, вопросы оптимизации серверной инфраструктуры становятся всё более актуальными. Одной из ключевых задач является разработка эффективных серверов для отдачи статического содержимого, которые обеспечивают быструю обработку запросов и минимизируют задержки в работе. Такие серверы широко применяются в веб-приложениях, онлайн-сервисах и системах доставки контента, что делает их незаменимыми элементами современной инфраструктуры.

При проектировании серверов для обработки статического содержимого особое внимание уделяется выбору архитектурного подхода. Существуют различные варианты организации работы сервера, среди которых различные комбинации thread pool, prefork и select, pselect, poll, epoll.

Каждая из перечисленных архитектур имеет свои особенности, преимущества и недостатки. Например, использование thread pool позволяет эффективно распределять нагрузку на ресурсы сервера за счёт заранее подготовленного пула потоков, тогда как подход prefork предполагает создание нескольких процессов для параллельной обработки запросов. С другой стороны, различия между механизмами управления событиями, такими как select, pselect, poll и epoll, заключаются в их способности обрабатывать множество одновременных соединений. Среди них epoll является наиболее современным и производительным решением, особенно для высоконагруженных систем.

Подход thread pool + epoll сочетает в себе преимущества предварительного создания потоков и эффективного механизма обработки событий. Это позволяет минимизировать задержки, сократить затраты на создание потоков или процессов во время выполнения и обеспечить высокую производительность даже при большом количестве одновременных подключений.

Целью данной работы является реализация сервера для отдачи статического содержимого с диска с использованием архитектуры по варианту 7: **thread pool + epoll** — с поддержкой запросов GET и HEAD и поддержкой статусов 200, 403, 404, 405, с учётом минимальных требований к безопасности серверов статического содержимого.

Для достижения поставленной цели необходимо решить следующие

задачи:

— провести анализ предметной области и изучить существующие подходы к построению серверов статического содержимого;

— спроектировать архитектуру статического сервера по варианту 7: с использованием thread pool + epoll;

— реализовать сервер на основе спроектированной архитектуры с поддержкой запросов `GET` и `HEAD`, обработкой ошибок и корректной передачей файлов размером до 128 Мб;

— выполнить нагрузочное тестирование сервера

— проанализировать результаты тестирования и сравнить их с результатами Nginx.

1 Аналитическая раздел

1.1 Протокол TCP

Протокол TCP (Transmission Control Protocol) является одним из основных протоколов транспортного уровня в модели OSI и используется для обеспечения надёжной передачи данных между приложениями через сеть [1]. TCP обеспечивает:

— установление соединения (трёхэтапное рукопожатие);

— контроль целостности данных с помощью проверки контрольной суммы;

— управление потоком для предотвращения перегрузки сети;

— повторную отправку потерянных пакетов;

— гарантированное получение данных в правильном порядке [2].

Благодаря этим характеристикам TCP часто используется в веб-приложениях, где требуется надёжность передачи данных, включая работу HTTP-серверов.

1.2 Протокол HTTP

HTTP (HyperText Transfer Protocol) — это прикладной протокол, предназначенный для передачи гипертекстовых документов, таких как HTML-страницы [3]. HTTP является основой взаимодействия между клиентами (обычно веб-браузерами) и серверами.

Основные возможности протокола HTTP включают:

— передачу данных в формате клиент-сервер;

— поддержку текстовых и бинарных форматов данных;

— использование URI (Uniform Resource Identifier) для идентификации ресурсов.

HTTP поддерживает методы запросов, включая:

— `GET` — получение ресурса с сервера;

— `HEAD` — получение метаинформации о ресурсе (заголовки ответа без тела);

— `POST`, `PUT`, `DELETE` и другие для выполнения различных операций [3].

Для статического сервера ключевыми являются методы `GET` и `HEAD`, а также корректная обработка ошибок с возвращением кодов статуса, таких как `200` (OK), `403` (Доступ запрещён), `404` (Не найден) и `405` (Метод не поддерживается) [3]. Эти механизмы обеспечивают стабильную работу и информативность при взаимодействии клиента с сервером.

1.3 Сокеты

Сокет — это программный интерфейс для обмена данными между приложениями через сеть. Он представляет собой абстракцию конечной точки сетевого соединения и является универсальным механизмом для взаимодействия параллельных процессов, применяемым как в локальной системе, так и в распределенных системах [4]. Сокеты используются для установления соединений, передачи данных и обработки запросов. Для работы с TCP/IP-сетями сокеты предоставляют следующие основные операции:

— `socket()` — создание сокета;

— `bind()` — привязка сокета к определённому адресу и порту;

— `listen()` — перевод сокета в режим ожидания входящих соединений;

— `accept()` — принятие входящего соединения;

— `recv()` и `send()` — получение и отправка данных;

— `close()` — закрытие сокета [4].

Сокеты обеспечивают базовый механизм передачи данных, что делает их фундаментальным инструментом для реализации серверов и клиентских приложений. Схема взаимодействия представлена на рисунке 1.1.

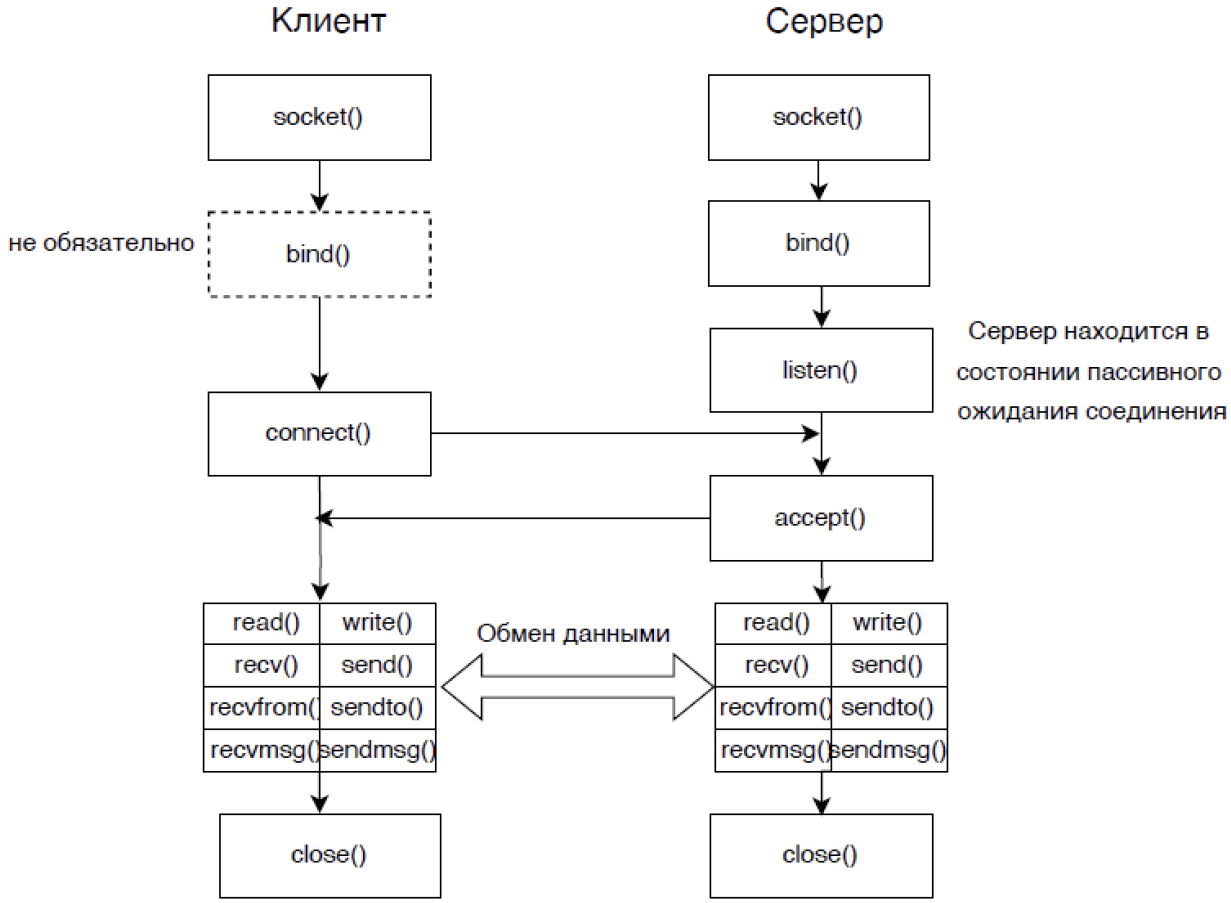

Рисунок 1.1 – Схема взаимодействия на сокетах по модели «клиент-сервер»

1.4 Статический сервер

Статический сервер — это сервер, предназначенный для обработки запросов на получение неизменяемого содержимого, такого как HTML-файлы, изображения, CSS-стили и JavaScript-файлы [5]. Задача статического сервера — предоставление клиентам запрашиваемого содержимого с минимальными задержками.

Для работы статического сервера важно обеспечить корректное:

— управление файловой системой для поиска и передачи запрошенных данных;

— обработку HTTP-запросов и формирование HTTP-ответов;

— запись логов для отслеживания событий и ошибок;

— управление безопасностью, предотвращая несанкционированный доступ к файлам [6].

1.5 Мультиплексирование

Для сокращения времени блокировки сервера при ожидании соединения используется мультиплексирование, поскольку время на установление связи с первым готовым к взаимодействию клиентом меньше, чем с каждым отдельным клиентом в определенной последовательности. Мультиплексор периодически проверяет состояние соединений, и первое доступное соединение выбирается ядром [4]. Мультиплексирование является менее ресурсоемким вариантом многозадачности. Операционная система предоставляет системные вызовы, с помощью которых можно использовать один из доступных мультиплексоров.

Основные методы мультиплексирования включают:

— `select()` — простой механизм, который позволяет отслеживать множество дескрипторов сокетов. Однако он имеет ограничения по числу отслеживаемых соединений и требует проверки каждого дескриптора на наличие события;

— `poll()` — улучшенная версия `select()`, не ограниченная количеством соединений, но сохраняющая проблемы масштабируемости;

— `pselect()` — усовершенствованная версия `select()`, которая позволяет задавать тайм-ауты с учетом сигналов, улучшая управление временем ожидания и возможность обработки сигналов в процессе ожидания;

— `epoll()` — современный механизм, предоставляющий события только для активных дескрипторов, что значительно повышает производительность в высоконагруженных системах [4].

На рисунке 1.2 представлена схема мультиплексирования.

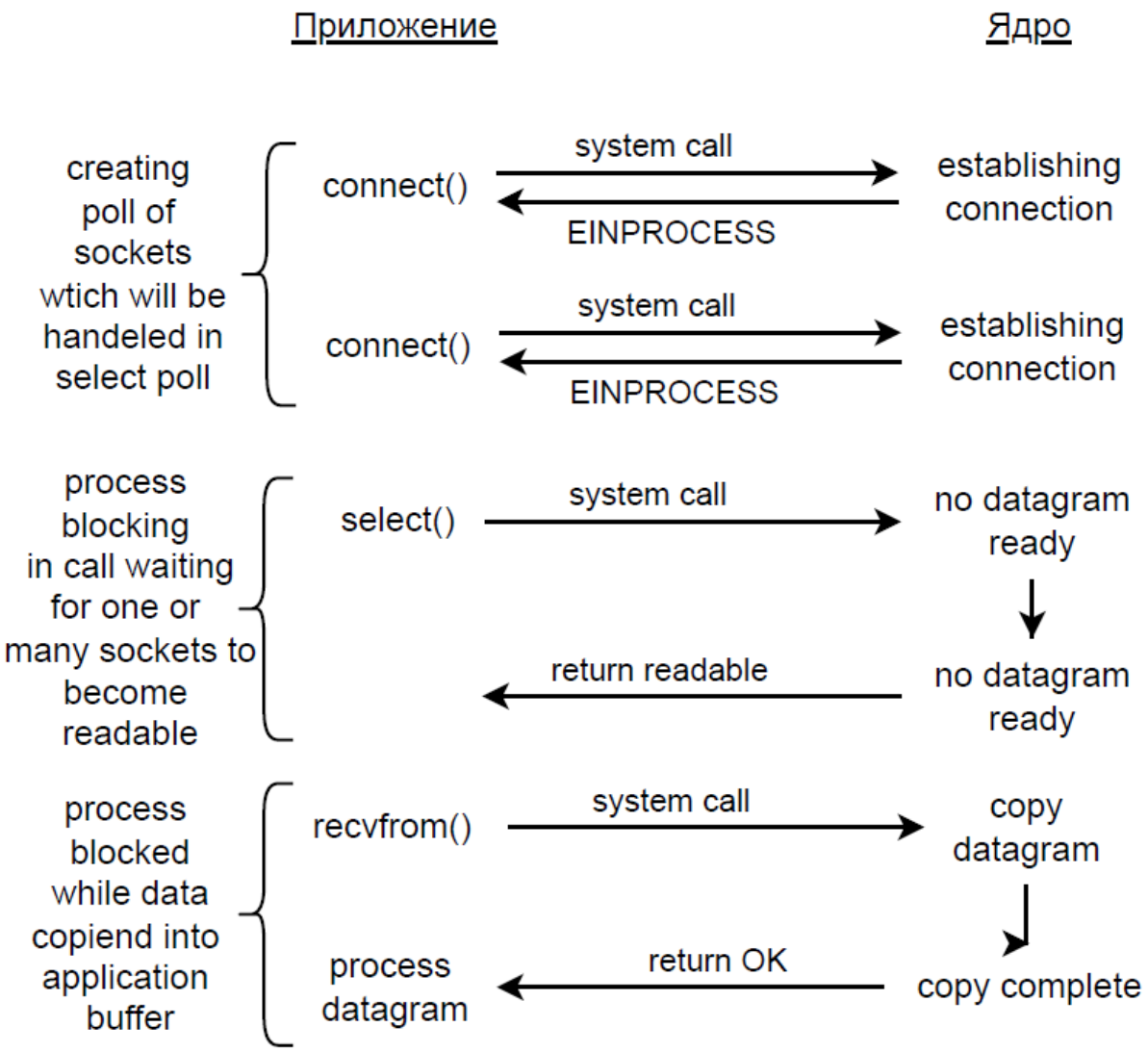

Рисунок 1.2 – Схема мультиплексирования.

1.6 Типы архитектур серверов

При проектировании серверов используются различные архитектуры, среди которых:

1. **Thread Pool (пул потоков)** — заранее создаётся фиксированное количество потоков, которые распределяются между задачами. Подход эффективен с точки зрения использования ресурсов;

2. **Prefork (многопроцессный)** — создаются отдельные процессы для обработки запросов. Такая архитектура обеспечивает изоляцию между запросами, но требует больше ресурсов;

3. **Событийно-ориентированная архитектура** — применяется механизм мультиплексирования (например, `epoll`) для обработки нескольких соединений в одном потоке [7].

Каждый подход имеет свои преимущества и недостатки. Для высоконагруженных систем предпочтительным является использование пула потоков в сочетании с `epoll`, что обеспечивает высокую производительность и минимальную задержку при большом количестве одновременных подключений.

Вывод

Таким образом, выбор архитектуры сервера и подхода к обработке запросов зависит от целевого сценария использования и требований к производительности и масштабируемости. В данном разделе был проведен анализ предметной области статического сервера и изучены типы существующих архитектур для статического сервера.

2 Конструкторский раздел

2.1 Схема работы статического сервера thread pool + epoll

Алгоритм работы статического сервера thread pool + epoll:

1. Сервер создаёт сокет и настраивает его для работы в неблокирующем режиме.

2. Сокет привязывается к порту и начинает прослушивание входящих соединений.

3. Сервер создаёт пул потоков, каждый из которых будет обрабатывать соединения с клиентами.

4. Сервер использует механизм `epoll` для эффективного мультиплексирования входящих соединений.

5. Сервер ожидает завершения работы всех потоков и закрывает сокет по завершении.

На рисунке 2.1 приведена схема работы статического сервера с thread pool + epoll.

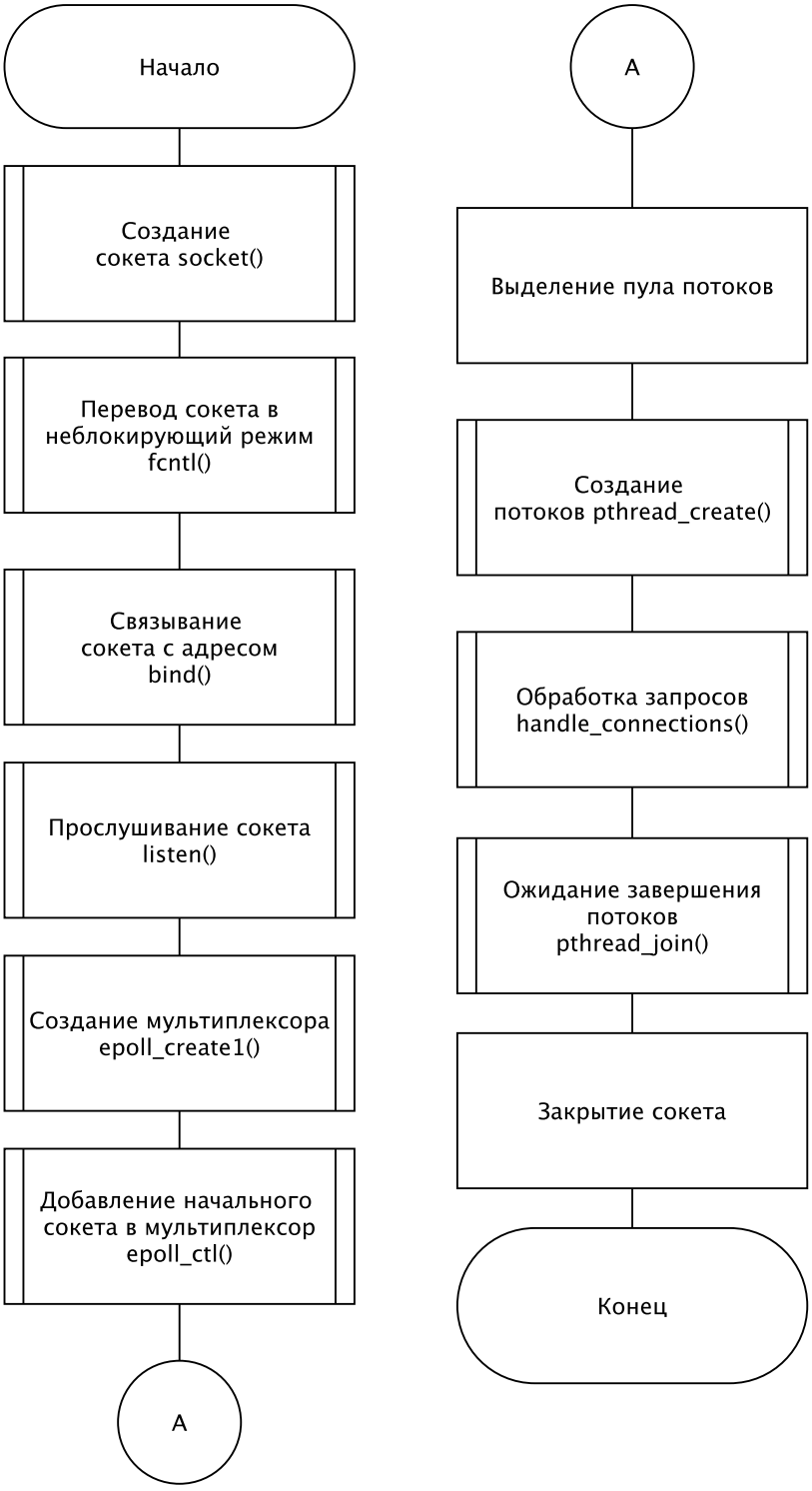

Рисунок 2.1 – Схема работы статического сервера с thread pool + epoll

Таким образом, сервер запускается, обрабатывает множество соединений одновременно с помощью многозадачности и эффективного механизма эполлинга и завершает свою работу, когда все задачи выполнены.

2.2 Схема работы потока с мультиплексированием

На рисунке 2.2 показана схема работы потока с мультиплексированием для статического сервера.

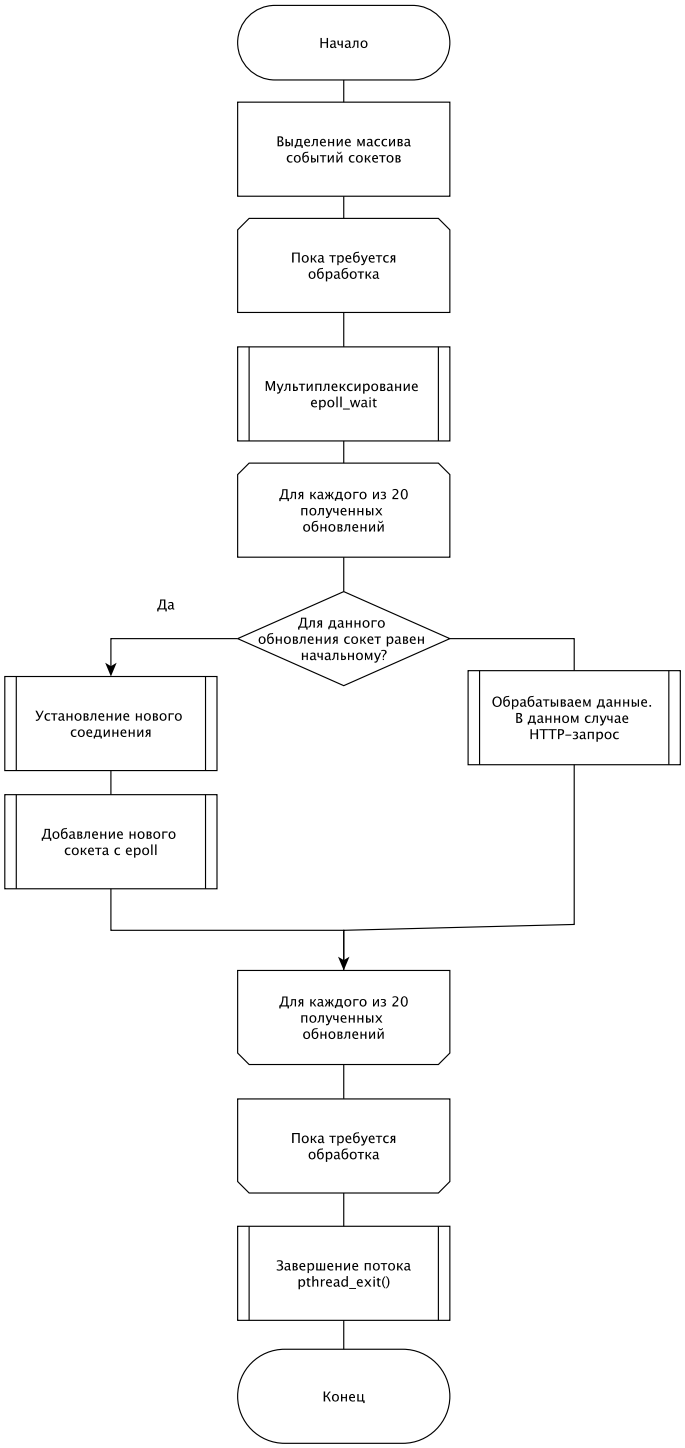

Рисунок 2.2 – Схема работы потока с мультиплексированием

Функция эффективно обрабатывает несколько соединений одновременно, используя механизмы мультиплексирования и многозадачности.

Вывод

В данном разделе была спроектирована архитектура статического сервера по варианту 7: с использованием thread pool + epoll.

3 Технологическая раздел

В листингах 3.1 - 3.3 приведена реализация статического сервера thread pool + epoll.

Листинг 3.1 – Листинг реализации статического сервера.

```
1  struct sockaddr_in sa;
2  sockfd = socket(AF_INET, SOCK_STREAM, 0);
3  ...
4  if (bind(sockfd, (struct sockaddr *) &sa, sizeof(sa)) == -1) {
5      log_fatal("bind socket server failed");
6  }
7  ...
8  listen(sockfd, CLIENT_NUM);
9  ...
10 epollfd = epoll_create1(0);
11 ...
12 if (epoll_ctl(epollfd, EPOLL_CTL_ADD, sockfd, &e) < 0) {
13     log_fatal("epoll_ctl failed for server socket");
14 }
15 ...
16 for (int i = 0; i < SIZE_POOL; i++) {
17     if (pthread_create(&thread_pool[i], NULL,
          handle_connections, NULL) != 0) {
18         log_fatal("pthread_create %d failed", i);
19     }
20 }
21 handle_connections(NULL);
22 for (int i = 0; i < SIZE_POOL; i++) {
23     pthread_join(thread_pool[i], NULL);
24 }
25
26 close(sockfd);
```

При добавлении нового соединения вызывается *epoll_ctl* с параметром $EPOLL\_CTL\_ADD$ для создания клиентского сокета ожидающего обработку.

Реализована попеременная обработка клиентов при передачи объёмных файлов. Для этого в *handle_connections* при обработке ожидаемых событий учитываются их стадии обработки: CONNECT, SEND, COMPLETE, ERROR. Если данные не были переданы полностью и необходимо продол-

жить отправку вызывается *epoll_ctl* для клиентского сокета с параметром $EPOLL\_CTL\_MOD$, чтобы пометить соединение как вновь ожидающее обработку.

После завершения обработки соединения вызывается *epoll_ctl* для клиентского сокета с параметром $EPOLL\_CTL\_DEL$ для удаления сокета.

Листинг 3.2 – Листинг мультиплескирования.

```
while (running) {
    int n = epoll_wait(epollfd, events, MAX_EVENTS, -1);
    if (n < 0) {
        if (!running) {
            break;
        }
        log_fatal("epoll_wait failed");
    }
    for (int i = 0; i < n; i++) {
        if (events[i].data.fd == sockfd) {
            conn_accept();
            continue;
        }

        int fd = events[i].data.fd;
        client_request_t *request = &clients[fd];

        switch (request->state) {
            case STATE_CONNECT:
                read_request(request);
            case STATE_SEND:
                http_handle(request);
                break;
            case STATE_COMPLETE:
                epoll_ctl(epollfd, EPOLL_CTL_DEL, fd, NULL);
                close(fd);
                log_info("Close connection correct");
                request->socket = -1;
                break;
            case STATE_ERROR:
                epoll_ctl(epollfd, EPOLL_CTL_DEL, fd, NULL);
                close(fd);
                log_info("Close connection with error");
```

```
34                  request->socket = -1;
35                  break;
36          }
37          if (request->state < STATE_COMPLETE) {
38              struct epoll_event e;
39              e.events = EPOLLIN | EPOLLET | EPOLLOUT;
40              e.data.fd = fd;
41              if (epoll_ctl(epollfd, EPOLL_CTL_MOD, fd, &e) < 0) {
42                  log_fatal("epoll_ctl modify failed");
43              }
44          }
45      }
46  }
```

Для обработки соединения используется структура *client_request_t* с полями сокет, стадия обработки, количество прочитанных байтов, количество переданных байтов, запрос.

Листинг 3.3 – Листинг передачи данных.

```
1  int http_copy_segment(int dst, int src, client_request_t
       *request) {
2      char buf[PAGE_SIZE] = {0};
3      ssize_t n = pread(src, buf, PAGE_SIZE, request->bytes_read);
4      if (n < 0) {
5          return ERR_READ;
6      }
7      if (n == 0) {
8          return ERR_EOF;
9      }
10     ssize_t written = write(dst, buf, n);
11     if (written < 0) {
12         return ERR_WRITE;
13     }
14     request->bytes_read += n;
15     request->bytes_to_send -= written;
16     if (n < PAGE_SIZE){
17         request->state = STATE_COMPLETE;
18     }
19     return OK;
20 }
```

4 Исследовательский раздел

На рисунке 4.1 показан пример работы сервера для отдачи статического содержимого с диска.

```
  # curl -v -X GET http://localhost:8080/1.txt --output
Note: Unnecessary use of -X or --request, GET is already inferred.
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /1.txt HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.88.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 17
< Content-Type: text/plain; charset=us-ascii
<
* Connection #0 to host localhost left intact
```

Рисунок 4.1 – GET запрос.

Было проведено нагрузочное тестирование реализованного сервера для отдачи статического содержимого с диска в сравнении с Nginx [8] для исследования пропускной способности. Тестирование проведено с использованием ApacheBenchmark [9], 2500 запросов картинки 3 Мб. Результаты нагрузочного тестирования представлены в таблице 4.1.

Технические характеристики устройства, на котором проводилось исследование:

— процессор: AMD Ryzen 5 4600H with Radeon Graphics [10];

— оперативная память: 16 Гбайт;

— операционная система: Windows 11, 64-битная, версия 23H2.

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями. Оба сервера были подняты в контейнере Docker [11].

Таблица 4.1 – Количества запросов, обрабатываемых сервером в секунду

Количество потоков	**Nginx**	**Сервер с thread pool + epoll**
1	50	37
2	51	44
4	52	53
8	51	58
16	48	64

По таблице видно, что NGINX скорость обработки Nginx снижается при увеличении количества потоков, уже при 4 потоках сервер с thread pool + epoll показывает лучшие результаты.

ЗАКЛЮЧЕНИЕ

Целью данной работы бфла достигнута: реализован сервер для отдачи статического содержимого с диска с использованием архитектуры по варианту 7: **thread pool + epoll** — с поддержкой запросов GET и HEAD и поддержкой статусов 200, 403, 404, 405, с учётом минимальных требований к безопасности серверов статического содержимого.

Для достижения поставленной цели были решены следующие задачи:

— проведен анализ предметной области и изучены существующие подходы к построению серверов статического содержимого;

— спроектирована архитектура статического сервера по варианту 7: с использованием thread pool + epoll;

— реализован сервер на основе спроектированной архитектуры с поддержкой запросов `GET` и `HEAD`, обработкой ошибок и корректной передачей файлов размером до 128 Мб;

— выполнено нагрузочное тестирование сервера

— проанализированы результаты тестирования и проведено сравнение их с результатами Nginx.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Лейкин А.* Протоколы транспортного уровня UDP, TCP и SCTP: достоинства и недостатки // Первая миля. — 2013. — Т. 38, № 5. — С. 62—69.
2. *Eddy W.* RFC 9293: Transmission Control Protocol (TCP). — 2022.
3. *Черненко Д.*, *Гниденко А.*, *Панкевич Д.* Исследование протокола HTTP // Материалы докладов в 53-й международной научно-технической конференции преподавателей и студентов. — 2020. — С. 32—33.
4. *Рязанова Н. Ю.*, Конспект курса лекций «Операционные системы» МГТУ им. Н .Э. Баумана, 2023 — 2024 гг.
5. *Li P.*, *Zdancewic S.* Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-Level Concurrency Primitives. — 2007.
6. *Елизаров О. О.* АВТОМАТИЗИРОВАННАЯ СИСТЕМА ОБРАБОТКИ СТАТИЧЕСКИХ ФАЙЛОВ // Инновации. Наука. Образование. — 2021. — № 34. — С. 1423—1435.
7. *Рязанов О. Ю.*, *Федотов Е. А.*, *Поляков В. М.* HTTP сервер с использованием пула потоков. — 2017.
8. *DeJonghe D.* Nginx CookBook. — O'Reilly Media, 2020.
9. Apache HTTP Server Version 2.4 [Электронный ресурс]. — — Режим доступа: `https://httpd.apache.org/docs` (дата обращения: 01.11.2024).
10. AMD Ryzen 5 4600H with Radeon Graphics [Электронный ресурс]. — — Режим доступа: `https://www.amd.com/en/products/apu/amd-ryzen-5-4600h` (дата обращения: 01.10.2024).
11. Docker [Электронный ресурс]. Режим доступа: https://docker.com (дата обращения: 10.12.2024).

ПРИЛОЖЕНИЕ А

Презентация содержит 9 слайдов.