



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 классического статического сервера для отдачи  
контента с диска»

Студент группы ИУ7-75Б

\_\_\_\_\_  
(Подпись, дата)

**П. А. Иванов**

\_\_\_\_\_  
(И.О. Фамилия)

Руководитель курсовой работы

\_\_\_\_\_  
(Подпись, дата)

**Н.О. Яковидис**

\_\_\_\_\_  
(И.О. Фамилия)

2023 г.

## РЕФЕРАТ

Расчетно-пояснительная записка 37 с., 12 рис., 0 табл., 24 ист., 0 прил.

МНОГОПОЛЬЗОВАТЕЛЬСКИЙ СЕРВЕР, МУЛЬТИПЛЕКСИРОВАНИЕ,  
LINUX, СОКЕТЫ, EPOLL, PRE-FORKING, КЛИЕНТ-СЕРВЕР

Объектом исследования данной работы является многопользовательский сервер.

Целью работы является разработка классического статического сервера для отдачи контента с диска на основе технологий pre-fork и epoll.

Поставленная цель достигается путем анализа способов проектирования многопользовательских серверов, проектированием статического сервера на основе архитектуры pre-fork с использованием epoll, реализацией сервера. Кроме того, проведено сравнение результатов нагрузочного тестирования разработанного сервера с nginx.

# СОДЕРЖАНИЕ

|   |           |
|---|-----------|
| <b>РЕФЕРАТ</b>  | <b>2</b>  |
| <b>ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ</b>                    | <b>4</b>  |
| <b>ВВЕДЕНИЕ</b>   | <b>5</b>  |
| <b>1 Аналитический раздел</b>                                   | <b>6</b>  |
| 1.1 Клиент-серверная архитектура . . . . .                      | 6         |
| 1.2 Сокеты . . . . .  | 6         |
| 1.2.1 Передача данных . . . . .                                 | 8         |
| 1.3 Мультиплексирование . . . . .                               | 9         |
| 1.3.1 select . . . . .  | 9         |
| 1.3.2 pselect . . . . .   | 10        |
| 1.3.3 poll . . . . .  | 11        |
| 1.3.4 epoll . . . . .   | 11        |
| 1.4 Модели конкурентных серверов . . . . .                      | 12        |
| 1.5 Проблема «громоподобного стада» . . . . .                   | 14        |
| <b>2 Конструкторский раздел</b>                                 | <b>16</b> |
| 2.1 Схемы алгоритмов . . . . .                                  | 16        |
| 2.2 Проектирование компонентов системы . . . . .                | 20        |
| <b>3 Технологический раздел</b>                                 | <b>22</b> |
| 3.1 Средства реализации системы . . . . .                       | 22        |
| 3.1.1 Выбор языка программирования и среды разработки . . . . . | 22        |
| 3.2 Реализация основных модулей . . . . .                       | 22        |
| <b>4 Исследовательский раздел</b>                               | <b>30</b> |
| 4.1 Технические характеристики . . . . .                        | 30        |
| 4.2 Подготовка исследования . . . . .                           | 30        |
| 4.3 Результаты замеров . . . . .                                | 32        |
| <b>ЗАКЛЮЧЕНИЕ</b>   | <b>34</b> |
| <b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>                         | <b>35</b> |

## **ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ**

API – Application Programming Interface [1];

Файловый дескриптор – возвращенное ядром системы целое неотрицательное число, предназначенное для работы с файлом [4].

## ВВЕДЕНИЕ

Современные сервера должны уметь обрабатывать запросы многих пользователей одновременно. Как правило, для этого применяют три основные техники: мультиплексирование, создание новых процессов (англ. *forking*) и создание новых потоков (англ. *threading*). Существуют также модификации последних: *pre-forking* и *pre-threading*, идея которых заключается в уменьшении времени задержки ответа путем использования пула процессов или потоков соответственно [2]. Такие сервера, как правило, работают на сокетах – абстракции конечных точек соединения, для работы с которыми в Unix предусмотрены такие API как *select*, *pselect*, *poll* и *epoll* [3].

Целью данной курсовой работы является разработка классического статического сервера для отдачи контента с диска на основе технологий *pre-fork* и *epoll*.

Для достижения поставленной цели необходимо решить следующие задачи:

- проанализировать способы проектирования многопользовательских серверов, изучить предоставляемые Unix-системами API для их создания;
- спроектировать статический сервер на основе архитектуры *pre-fork* с использованием *epoll*;
- реализовать сервер и протестировать разработанное программное обеспечение;
- провести сравнение результатов нагрузочного тестирования разработанного сервера с *nginx*.

# 1 Аналитический раздел

## 1.1 Клиент-серверная архитектура

Клиент-серверная архитектура является фундаментальным понятием, используемым для организации взаимодействия процессов (возможно, запущенных на разных вычислительных машинах) [2].

Сервер – процесс, способный принимать запросы, обрабатывать их и, при наличии и необходимости, возвращать ответ. Клиент – процесс, который отправляет запросы. Таким образом, сервер предоставляет «услуги», а клиент их запрашивает, инициируя коммуникацию.

Модель клиент-серверной архитектуры представлена на рисунке 1.

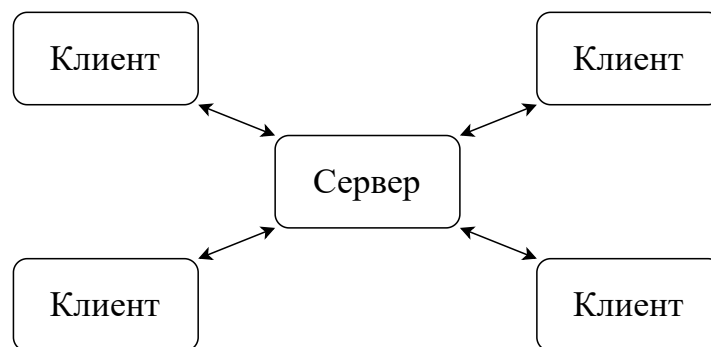


Рисунок 1 – Модель клиент-серверной архитектуры

## 1.2 Сокеты

Сокет – абстракция конечной точки соединения (взаимодействия). Они делают возможной коммуникацию процессов как на одной, так и на разных вычислительных машинах [4].

Подобно тому как для работы с файлами приложения используют дескрипторы файлов, для работы с сокетами они используют дескрипторы сокетов.

В UNIX дескрипторы сокетов реализованы так же, как дескрипторы файлов. Разница между ними заключается в множестве допустимых операций, применимых к этим файлам [4].

Для коммуникации процессов по сети используется понятие порта – конечной точки соединения, обозначаемой 16-битным целым числом. Порты используются для идентификации процесса, которому предназначается пакет. Определение процесса по номеру порта происходит в ядре операционной системы в ходе операции, называемой «демультиплексированием». Это определение успешно, если некоторый процесс «связал» свой сокет с данным портом.

Сокет создается системным вызовом `socket()`. Его прототип представлен в листинге 1 [5].

#### Листинг 1 – Прототип системного вызова `socket()`

```
1 #include <sys/socket.h>
2 int socket(int domain, int type, int protocol);
```

Аргумент `domain` определяет природу взаимодействия, включая формат адреса. Например, домен Интернета IPv4 определяется константой `AF_INET`, а домен UNIX – `AF_UNIX`. Тип сокета (аргумент `type`) определяет его характеристики взаимодействия. Например, тип `SOCK_DGRAM` не ориентирован на создание логического соединения, допускает отправку сообщений только фиксированной длины и не гарантирует доставку сообщений. Тип `SOCK_STREAM` ориентирован на создание логического соединения, упорядоченность передачи данных и гарантирует их доставку. В качестве протокола (аргумент `protocol`), как правило, указывается 0, что означает использование протокола по умолчанию. Так, для `SOCK_DGRAM` из домена `AF_INET` это UDP, а для `SOCK_STREAM` из домена `AF_INET` – TCP [5].

### 1.2.1 Передача данных

Взаимодействие с использованием сокетов осуществляется по модели клиент-сервер. Последовательность системных вызовов при использовании протокола с установлением соединения (TCP) представлена на рисунке 2 [2].

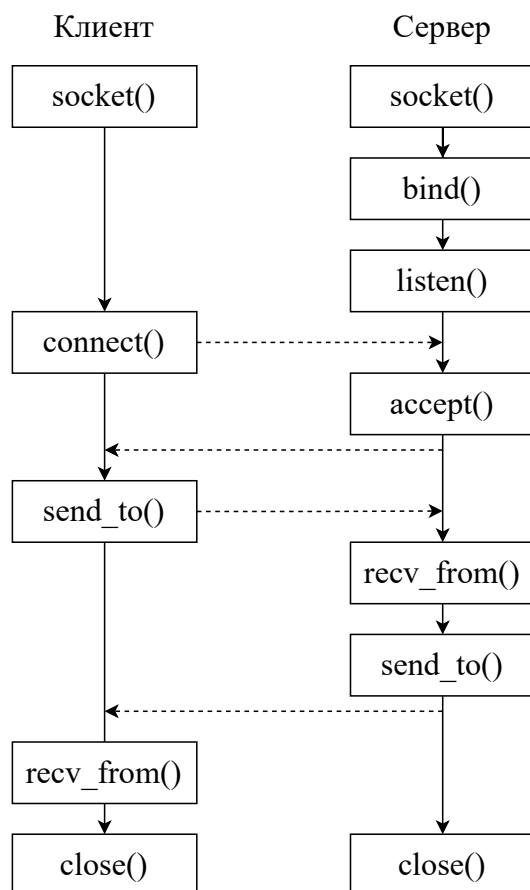


Рисунок 2 – Передача сообщения от клиента серверу

Системный вызов `bind()` [6] используется для назначения сокету локального адреса. Для сетевых сокетов определена структура `struct sockaddr_in` [4], в которой явно определяется порт и IP-адрес.

Вызов `connect()` [7] устанавливает соединение по адресу, который передается в функцию в качестве аргумента.

Когда сервер получил запрос на соединение, то будет выполнен системный вызов `accept()` – принять этот запрос [8]. Данный системный вызов создает копию исходного сокета и возвращает дескриптор файла нового сокета.



В результате исходный сокет остается в состоянии «listen», а копия – в состоянии «connected». Дублирование сокетов при принятии соединения дает возможность серверу принимать новые соединения без необходимости закрывать ранее принятые [4].

### 1.3 Мультиплексирование

Традиционный способ написания серверов – использовать главный поток, заблокированный на системном вызове `accept()` в ожидании новых подключений. Как только приходит новый запрос на соединение, сервер создает новый процесс системным вызовом `fork()` [9]. Дочерний процесс обрабатывает запрос, а родительский (главный) готов снова принимать запросы на подключение.

Чтобы избежать создания нового процесса под каждый запрос, что затратно по ресурсам, используют мультиплексирование запросов средствами API `select`, `poll`, `pselect` и `epoll`.

#### 1.3.1 `select`

Системный вызов `select()` позволяет программе отслеживать несколько файловых дескрипторов в ожидании, пока один или несколько из них станут «готовы» к какому-то виду операции ввода-вывода. Файловый дескриптор считается готовым, если соответствующую операцию ввода-вывода можно произвести без блокировки [10].

Но поскольку `select()` проектировался до появления концепции неблокирующего ввода-вывода, ряд трудностей делает его использование в современных системах нецелесообразным:

- Для выяснения того, какой именно дескриптор сгенерировал событие, необходимо вручную опросить их все с помощью `FD_ISSET`, что приводит к излишним затратам ресурсов;
- Максимальное количество одновременно наблюдаемых дескрипторов ограничено константой `FD_SETSIZE`, которая равна 1024;
- Закрывание дескриптора сокета, отслеживаемого API `select()`, не в главном потоке, приводит к неопределенному поведению;
- Невозможно динамически менять набор наблюдаемых событий;
- Отдельно необходимо вычислять наибольший дескриптор и передавать его отдельным параметром.

### 1.3.2 `pselect`

Как и `select()`, `pselect()` ждет изменения статуса нескольких файловых дескрипторов. Эти функции идентичны, за исключением трех отличий между ними [13]:

- 1) Функция `select` использует время ожидания, которое задано в структуре `struct timeval` (с секундами и микросекундами), тогда как `pselect` использует `struct timespec` (с секундами и наносекундами).
- 2) Функция `select` может обновить параметр `timeout`, который показывает сколько времени прошло. Функция `pselect` не изменяет этот параметр.
- 3) Функция `select` не имеет параметра `sigmask`, и т.о. ведет себя также как функция `pselect` вызванная с этим параметром, установленным в `NULL`.

### 1.3.3 poll

После появления необходимости писать высоконагруженные сервера был спроектирован API `poll()` [11], учитывающий недостатки `select()`:

- Ограничение в 1024 файловых дескриптора отсутствует;
- Наблюдаемые структуры лучше структурированы;

Однако и `poll()` не лишен недостатков:

- Как и при использовании `select()`, невозможно определить какие именно дескрипторы сгенерировали события без полного прохода по всем наблюдаемым структурам и проверки в них поля `revents`;
- Как и при использовании `select()`, нет возможности динамически менять наблюдаемый набор событий.

### 1.3.4 epoll

Данный API появился как логическое продолжение `select()` и `poll()`. От них он отличается тем, что позволяет добавлять, удалять и модифицировать дескрипторы и события в наблюдаемом списке [12].

Последовательность работы с `epoll` следующая.

- 1) Создать дескриптор `epoll` с помощью вызова `epoll_create()`;
- 2) Инициализировать структуру `epoll_event` нужными событиями и указателями на контексты соединений;
- 3) Вызвать `epoll_ctl()` [14] посредством макроса `EPOLL_CTL_ADD` для добавления дескриптора в список наблюдаемых;
- 4) Вызвать `epoll_wait()` [15] для ожидания событий с указанием максимального числа событий, которое можно получить за раз;
- 5) Обработать полученные события.

Важным отличием является отсутствие необходимости просматривать весь список отслеживаемых дескрипторов.

Достоинства `epoll` следующие:

- Нет необходимости просматривать полный список структур в поисках той, возможно, одной, где сработало ожидаемое событие.
- Есть возможность добавлять или удалять сокеты из списка в любое время. Также можно модифицировать наблюдаемые события.
- Можно завести сразу несколько потоков, ожидающих события из одной и той же очереди с помощью `epoll_wait()`.

Несмотря на описанные улучшения, в некоторых ситуациях использование `epoll` нецелесообразно. Недостатки `epoll` следующие:

- Изменение флагов событий происходит средствами лишнего системного вызова `epoll_ctl()`, что добавляет лишнее переключение контекста;
- Для каждого нового соединения необходимо вызвать `accept()` и `epoll_ctl()` – это два системных вызова. В случае использования `poll` вызов будет лишь один. При очень коротком времени жизни соединения переключения контекста могут значительно понизить производительность.
- Отсутствие переносимости на платформы, отличные от Linux;

## 1.4 Модели конкурентных серверов

Самая простая модель обработки запросов к серверу – итеративная. В данной модели сервер может обрабатывать только одного клиента в единицу времени. Остальные клиенты блокируются до тех пор, пока не подойдет их очередь и им не будет возвращен ответ [2].

Для обработки нескольких запросов одновременно можно использовать следующий подход: при получении запроса на подключение для его обработки выделяется новый поток или процесс. При высокой нагрузке сервер, использу-

ющий данную модель, неэффективен, поскольку, при увеличении числа потоков или процессов, в мультизадачных системах кванты времени для обработки запросов начинают выделяться реже [2].

Чтобы избежать падения производительности сервера, связанного с затратами ресурсов на создания потока или процесса, используют следующие модели: pre-forking и pre-threading [2].

Pre-forking – создание пула копий процесса-родителя. Запрос обрабатывается любым свободным дочерним процессом. Таким образом исключаются затраты на создание процесса под каждый запрос. При повышении нагрузки процесс-родитель может увеличить размер пула. Данная модель представлена на рисунке 3 [2].

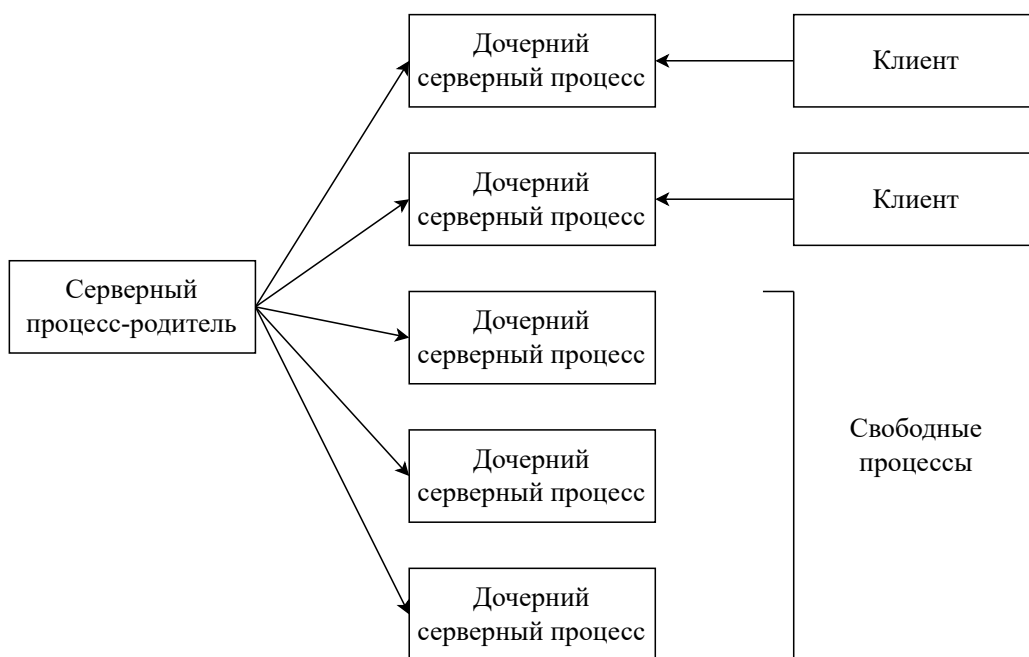


Рисунок 3 – Модель pre-forking

Pre-threading – создание пула потоков-обработчиков запросов. Идея данной модели аналогична pre-forking, только вместо пула процессов используется пул потоков (англ. thread pool). Данная модель представлена на рисунке 4 [2].

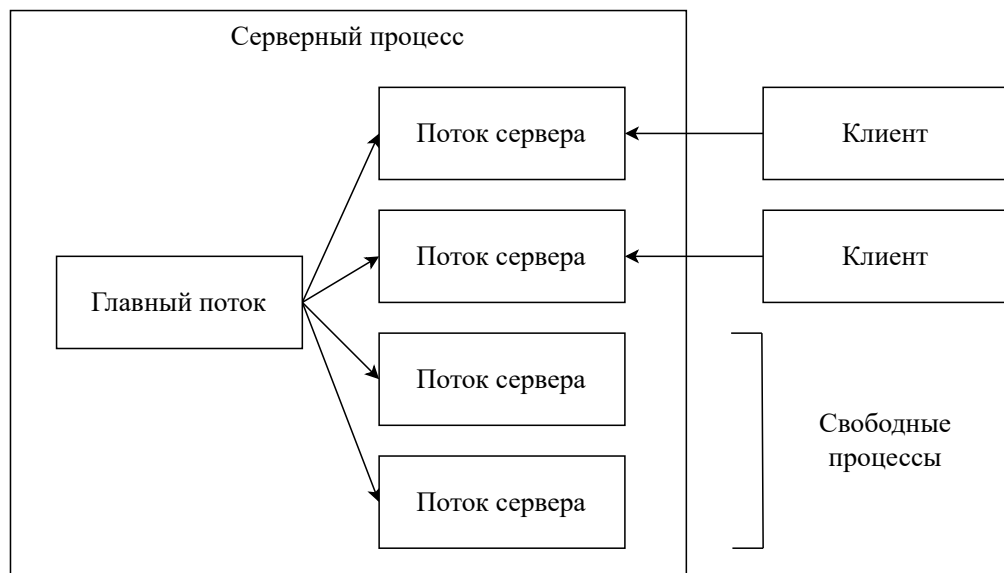


Рисунок 4 – Модель pre-threading

### 1.5 Проблема «громоподобного стада»

Проблема «громоподобного стада» (англ. *thundering herd problem*) ставится следующим образом. Имеется  $M$  дочерних процессов (или потоков). При старте сервера все из них вызывают `accept()` и блокируются. При первом запросе на подключение все  $M$  процессов (потоков) просыпаются. Только один из них перейдет к выполнению задания на обработки, а остальные будут вынуждены снова заблокироваться [2].

В случае `select` с этой проблемой ничего не поделать. В случае `epoll` данную проблему попытались решить флагом `EPOLLEXCLUSIVE`, который устанавливается вызовом `epoll_ctl` [14]. Данный флаг устанавливает монопольный режим пробуждения для файлового дескриптора `epoll`, ассоциируемого с файловым дескриптором `fd`. Когда данный флаг установлен, один или несколько файловых дескрипторов `epoll` получают уведомление о событии на `epoll_wait`. Поведение по умолчанию (без флага) – оповещение о событии вообще всех файловых дескрипторов.

## **Вывод**

В данном разделе были проанализированы способы проектирования многопользовательских серверов. Были рассмотрены 4 модели, две из которых признаны наиболее эффективными: pre-threading и pre-forking.

Также были рассмотрены основы сокетов и сетевого стека Linux, а также инструменты мультиплексирования: select, pselect, poll, epoll. Были приведены достоинства и недостатки каждого API, а также рассмотрена проблема «громоподобного стада» и метод её решения в случае epoll.

## 2 Конструкторский раздел

### 2.1 Схемы алгоритмов

На рисунке 5 представлен основной алгоритм работы родительского процесса сервера.

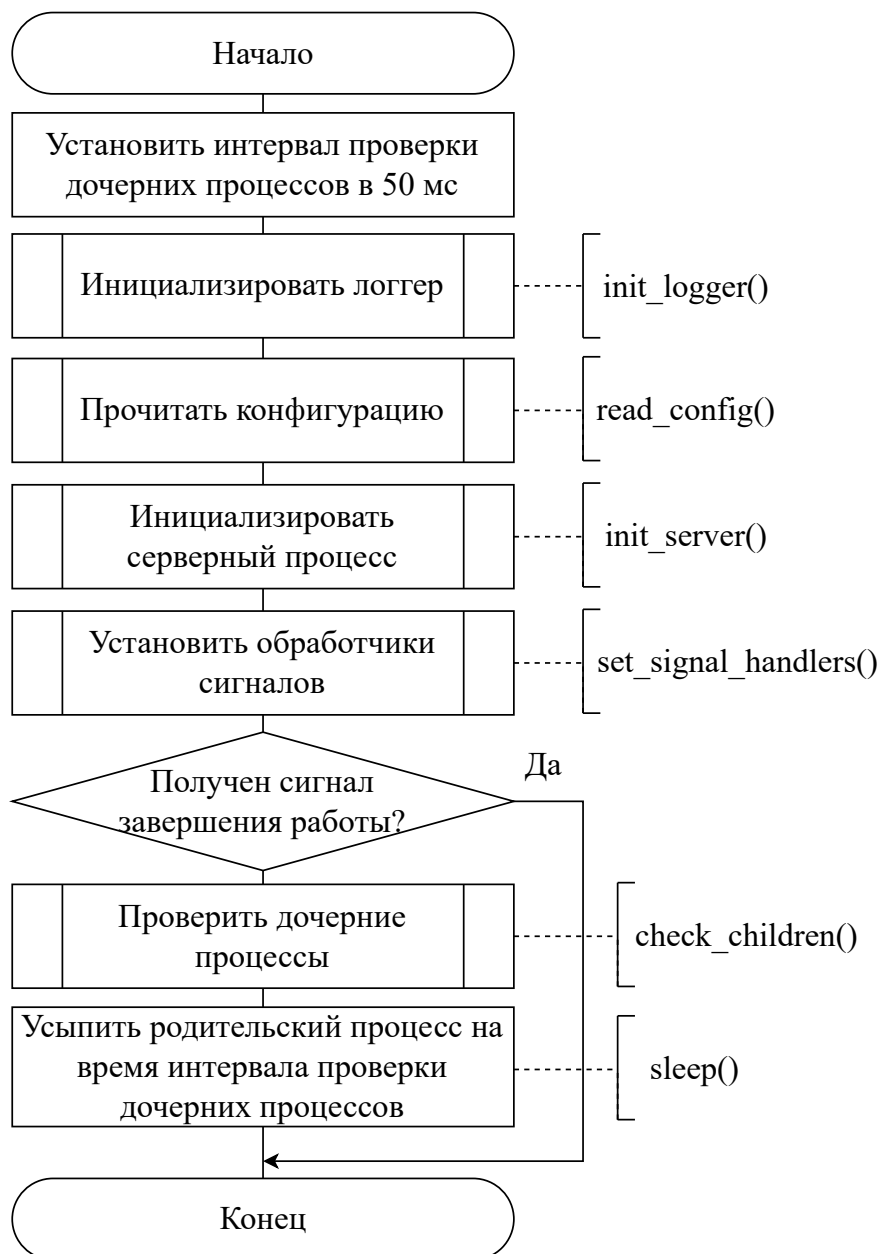


Рисунок 5 – Алгоритм работы процесса-родителя



На рисунке 6 представлен алгоритм функции инициализации сервера.

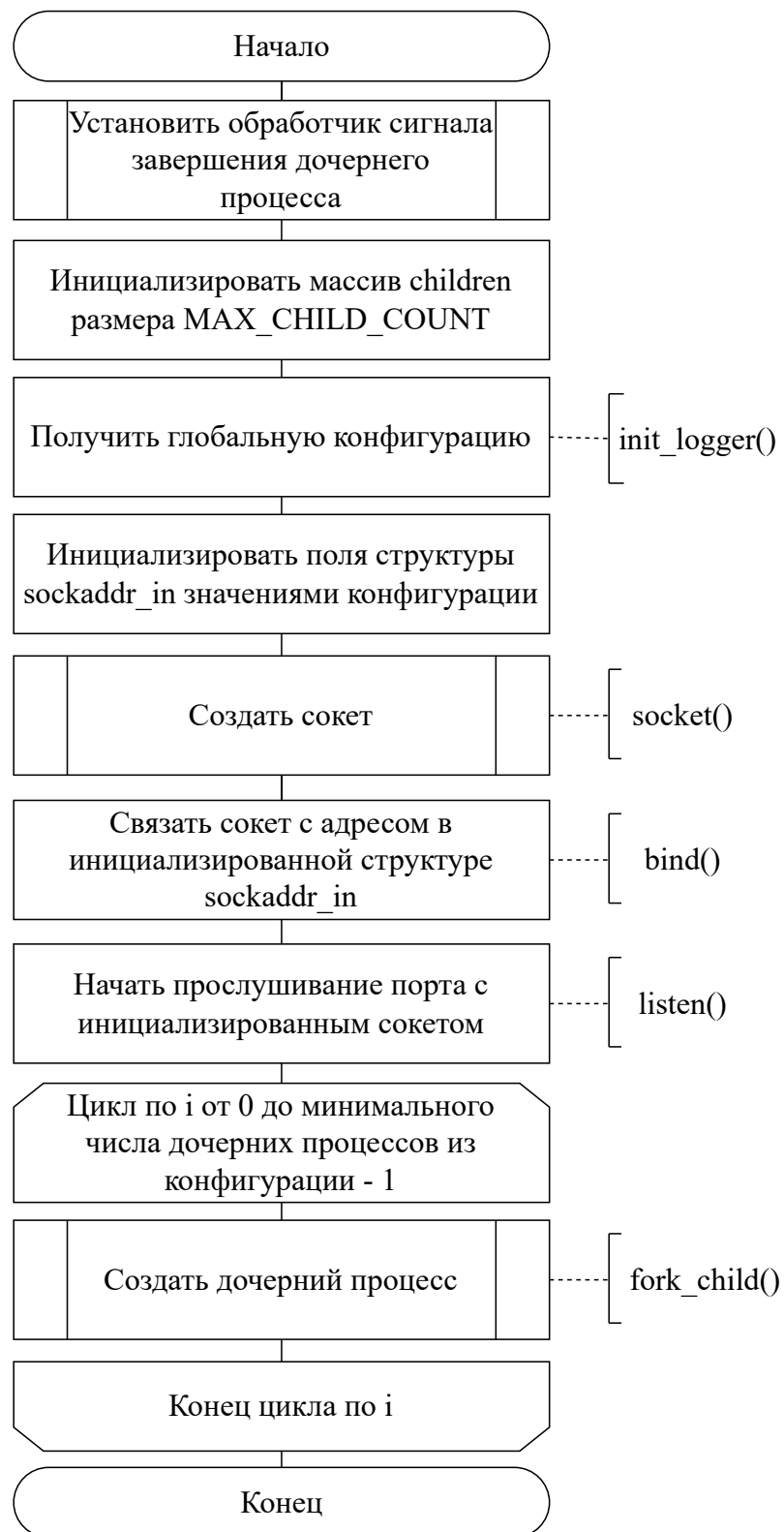


Рисунок 6 – Алгоритм функции init\_server()

На рисунке 7 представлен алгоритм создания дочернего процесса сервера.

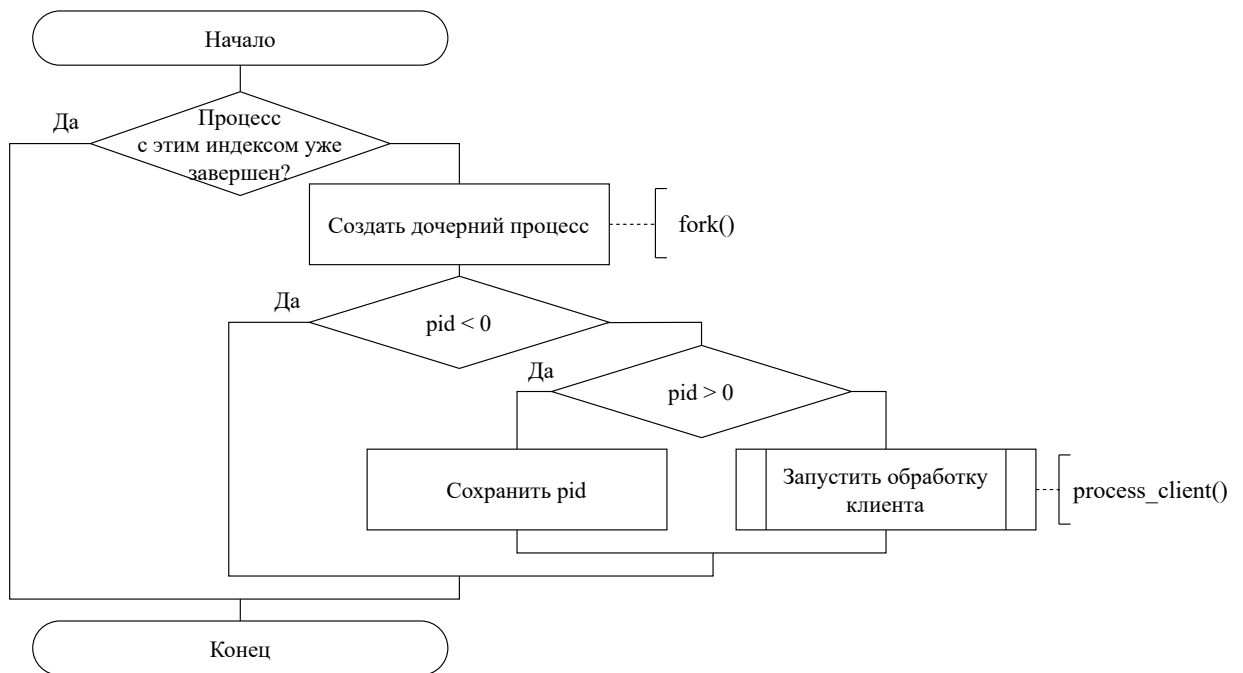


Рисунок 7 – Алгоритм функции `fork_child()`

На рисунке 8 представлен алгоритм функции инициализации дочернего процесса.

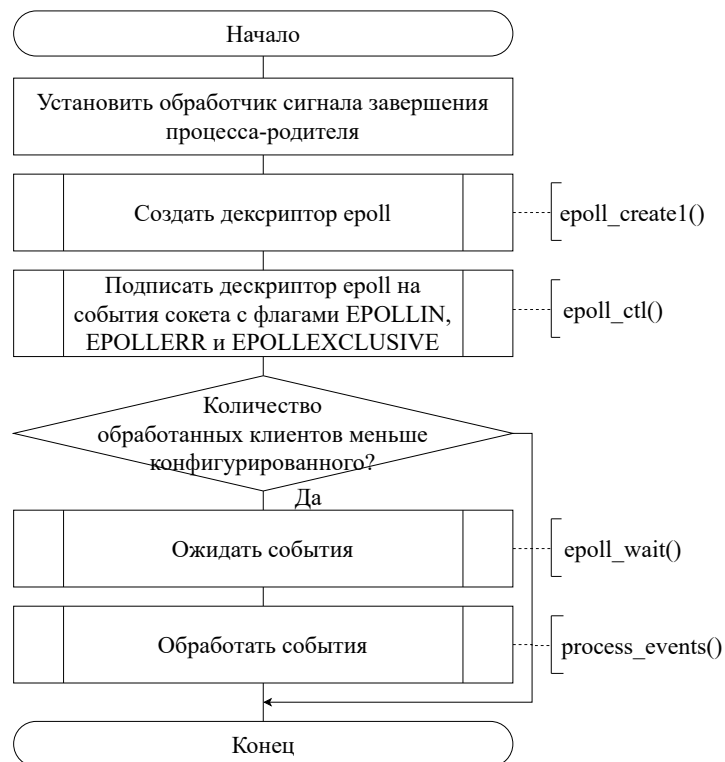


Рисунок 8 – Алгоритм функции `process_client()`

На рисунке 9 представлен алгоритм функции обработки события в виде запроса.



Рисунок 9 – Алгоритм функции `process_events()`

На рисунке 10 представлен алгоритм функции регулярной проверки пула процессов.



Рисунок 10 – Алгоритм функции `check_children()`

## 2.2 Проектирование компонентов системы

Для решения поставленной задачи спроектировано следующее разделение системы на программные компоненты:

- 1) `config` – модуль для работы с конфигурацией,
- 2) `http` – набор модулей для работы с заголовками HTTP,
- 3) `logger` – модуль для логирования происходящих в системе событий,
- 4) `prefork` – компонент, состоящий из модулей `parent` и `child`, реализующих

функциональность родительского и дочерних процессов,

5) `utils` – модуль вспомогательных функций,

6) `main` – точка входа в приложение.

### 3 Технологический раздел

#### 3.1 Средства реализации системы

##### 3.1.1 Выбор языка программирования и среды разработки

По условию задачи для реализации системы должен быть использован язык программирования C. В качестве стандарта языка выбран C99 [16], поскольку более новые стандарты не предоставляют дополнительных возможностей, которые можно было бы использовать в данной работе. Для компиляции кода выбран компилятор gcc [17].

В качестве среды разработки выбрана среда Visual Studio Code [18], поскольку она предоставляет все необходимые инструменты для написания и отладки кода на языке C.

#### 3.2 Реализация основных модулей

В листинге 2 представлен код точки входа в программу: сначала инициализируется логгер, затем читается конфигурация, инициализируется сервер и, наконец, выполняется цикл мониторинга дочерних процессов.

Листинг 2 – Точка входа в программу

```
1 int main()  
2 {  
3     init_logger("prefork.log");  
4  
5     read_config();  
6     init_server();
```

## Окончание листинга 2

```
7     set_signal_handlers();
8
9     while (1)
10    {
11        check_children();
12        usleep(CHILD_CHECK_INTERVAL_USEC);
13    }
14
15    return 0;
16 }
```

Код реализованной функции `init_server()` представлен в листинге 3.

## Листинг 3 – Реализация функции `init_server()`

```
1 void init_server()
2 {
3     struct sigaction sa;
4     memset(&sa, 0, sizeof(sa));
5     sa.sa_handler = sigchld_handler;
6     sigaction(SIGCHLD, &sa, NULL);
7
8     children = mmap(NULL, sizeof (server_item) *
9                     (MAX_CHILD_COUNT + 1),
10    PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
11
12     for(int i = 0; i < MAX_CHILD_COUNT; i++)
13     {
14         children[i].pid = 0;
15         children[i].state = SERVER_ITEM_DEAD;
16     }
17
18     config *config = config_get();
19     bind_server(config);
20 }
```

### Окончание листинга 3

```
19     int base_fork = config->min_children;
20     if (base_fork < 1 || base_fork > MAX_CHILD_COUNT)
21     {
22         die_with_error("min_children must be between 1 and
                MAX_CHILD_COUNT");
23     }
24
25     for (int i = 0; i < base_fork; i++)
26     {
27         fork_child(children + i);
28     }
29
30     used_children = base_fork;
31 }
```

Реализованная функция `fork_child()` представлен в листинге 4.

### Листинг 4 – Реализация функции `init_server()`

```
1 void fork_child(server_item *item)
2 {
3     if (item->state != SERVER_ITEM_DEAD)
4     {
5         return; // child is alive
6     }
7
8     item->state = SERVER_ITEM_AVAILABLE;
9
10    pid_t pid = fork();
11    if (pid < 0)
12    {
13        die_with_error("Fork failed");
14    }
15    else if (pid > 0)
16    {
```



#### Окончание листинга 4

```
17 // parent process
18 item->pid = pid;
19     }
20     else if (pid == 0)
21     {
22         // child process
23         process_client(server_socket, item);
24         exit(0);
25     }
26 }
27 }
```

Реализованная функция `process_client()` представлен в листинге 5.

#### Листинг 5 – Реализация функции `process_client()`

```
1 void process_client(int server_socket, server_item *item)
2 {
3     set_parent_death_signal();
4
5     config *config = config_get();
6
7     int epoll_fd = configure_epoll(server_socket);
8
9     start_processing_loop(epoll_fd, config, server_socket, item);
10 }
```

В листинге 6 представлена реализация функции `process_events()`.

#### Листинг 6 – Реализация функции `process_client()`

```
1 void process_events(struct epoll_event* events,
2     ssize_t events_count,
3     int server_socket,
4     server_item *item,
5     int* processed_clients)
6 {
```

## Продолжение листинга 6

```
7     for (int i = 0; i < events_count; i++)
8     {
9         struct sockaddr_storage client_addr;
10        socklen_t client_addrlen = sizeof client_addr;
11
12        if (events[0].events & EPOLLIN)
13        {
14            int fd = accept(server_socket, (struct sockaddr
15                             *)&client_addr, &client_addrlen);
16            if (fd < 0)
17            {
18                log(ERROR, "Server thread FAILED to proccess
19                    accept with EAGAIN");
20                continue;
21            }
22
23            item->state = SERVER_ITEM_BUSY;
24            http_parse_request* request = read_request(fd);
25
26            (*processed_clients)++;
27
28            if (check_request_errors(request, fd, item) > 0)
29                continue;
30
31            config_host *host;
32            if (!(host = get_host(fd, request, item)))
33                continue;
34
35            check_root_file(request);
36            remove_params(request->path);
37
38            log(INFO, "Returning file %s", request->path);
39            respond_file(fd, host, request);
```

#### Окончание листинга 6

```
38         END_CLIENT(item, fd, request);
39
40         item->state = SERVER_ITEM_AVAILABLE;
41
42         close(fd);
43     }
44 }
45 }
```

Реализация функции периодической проверки дочерних процессов представлена в листинг 7.

#### Листинг 7 – Реализация функции check\_children()

```
1 void check_children()
2 {
3     config *config = config_get();
4     int alive_count = 0, available_count = 0;
5
6     /*
7      * Get current children pool state
8      */
9     for(int i = 0; i < used_children; i++)
10    {
11        if (children[i].state == SERVER_ITEM_DEAD)
12        {
13            continue;
14        }
15
16        alive_count++;
17        if (children[i].state == SERVER_ITEM_AVAILABLE)
18        {
19            available_count++;
20        }
21    }
```

## Продолжение листинга 7

```
22     /*
23     * Count how many children should be added
24     */
25     int add_count = 0;
26     if (alive_count < config->min_children)
27     {
28         add_count = config->min_children - alive_count;
29     }
30     if (available_count == 0 &&
31         add_count == 0 &&
32         alive_count + 1 < config->max_children &&
33         alive_count + 1 < MAX_CHILD_COUNT)
34     {
35         add_count = 1;
36     }
37
38     /*
39     * Make dead children alive again
40     */
41     for (int i = 0; i < used_children && add_count > 0; i++)
42     {
43         if (children[i].state == SERVER_ITEM_DEAD)
44         {
45             fork_child(children + i);
46             add_count--;
47             available_count++;
48         }
49     }
50
51     /*
52     * Create new children
53     */
54     if (add_count > 0)
```

### Окончание листинга 7

```
55     {
56         for(int i = used_children; i < (used_children +
           add_count); i++)
57         {
58             fork_child(children + i);
59         }
60         used_children += add_count;
61     }
62 }
```

## 4 Исследовательский раздел

### 4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры, следующие.

- Операционная система Windows 10 [20] x86\_64.
- Память 8 Гб 2400 МГц DDR4.
- 1.6 ГГц 4-ядерный процессор Intel Core i5 8265U [21].

Замеры проводились на ноутбуке, включенном в сеть электропитания. Во время проведения исследований ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой выполнения замеров.

### 4.2 Подготовка исследования

Для сравнения производительности разработанного сервера с nginx необходимо установить утилиту для нагрузочного тестирования Apache Benchmark [22]. Данный инструмент позволяет исследовать допустимые границы количества запросов, которое сервер может обработать в секунду.

Также необходимо настроить nginx [23] на отдачу статического контента. Конфигурация nginx представлена в листинге 8.

Листинг 8 – Конфигурация nginx

```
1 worker_processes 5;
2 pid /var/run/nginx.pid;
3 error_log /var/log/nginx/error.log info;
4
5 events {
```

#### Окончание листинга 8

```
6   worker_connections  4096;
7 }
8
9 http {
10  charset utf-8;
11
12  server {
13      location / {
14          include /etc/nginx/mime.types;
15          # don't cache it
16          proxy_no_cache 1;
17          # even if cached, don't try to use it
18          proxy_cache_bypass 1;
19          root /static;
20      }
21
22      location /status {
23          stub_status;
24      }
25  }
26 }
```

Для запуска nginx используется Docker [19]. Содержимое используемого файла `docker-compose.yml` представлено в листинге 9.

#### Листинг 9 – Конфигурация `docker-compose.yml`

```
1 version: '3.7'
2 services:
3   nginx:
4     image: nginx
5     volumes:
6       - ./logs:/var/log/nginx
7       - ./nginx.conf:/etc/nginx/nginx.conf
8       - C:\Users\Pavel\Desktop\parcorpus\static:/static
```

#### Окончание листинга 9

```
9  ports:
10    - "80:80"
11    - "443:443"
12  restart:
13    always
```

### 4.3 Результаты замеров

Замеры проводились для двух файлов: для jpg-файла размером 53 Кб и pdf-файла размером 2.11 Мб. Были произведены замеры при постоянном числе запросов 10000 для jpg-файла и 1500 для pdf-файла и изменяющимся максимальным числом конкурентных запросов. На рисунках 11 и 12 представлена зависимость числа RPS (англ. requests per second – числа запросов в секунду) от числа конкурентных запросов для jpg и pdf файлов соответственно.

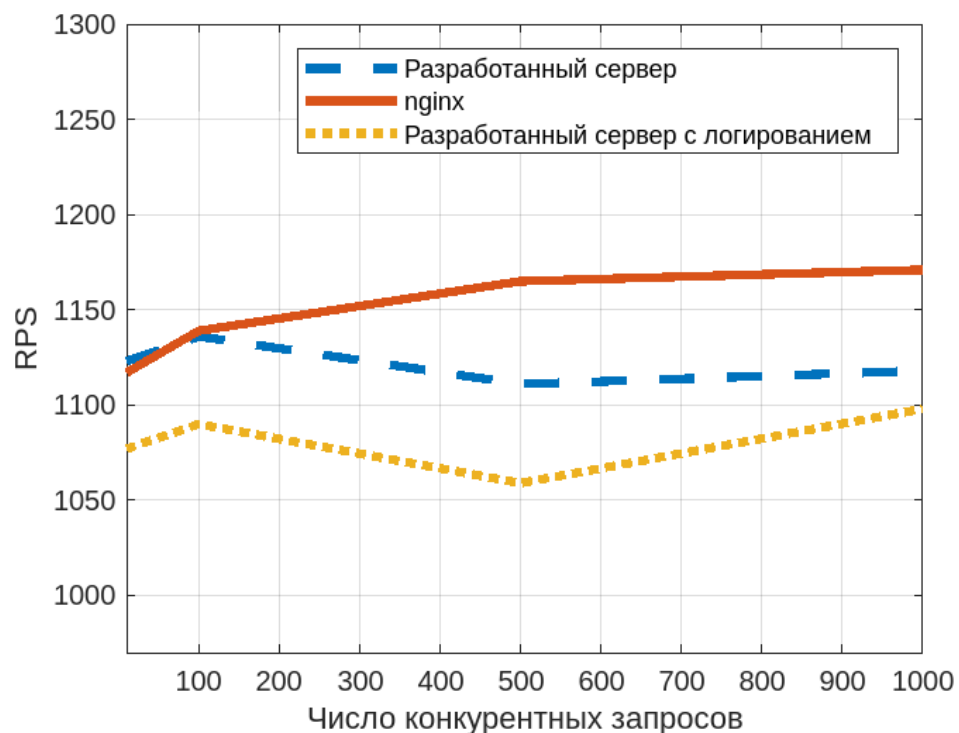


Рисунок 11 – Зависимость RPS от числа конкурентных запросов (jpg)



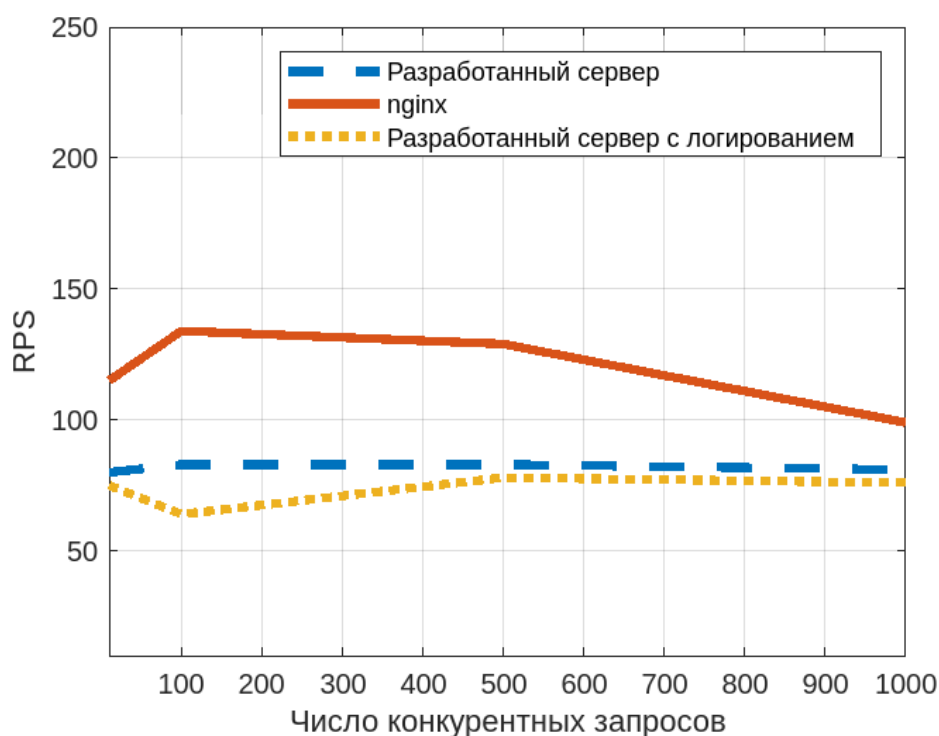


Рисунок 12 – Зависимость RPS от числа конкурентных запросов (pdf)

## Вывод

В результате проведенных замеров установлено, что разработанный сервер работает, как правило, медленнее, чем nginx. Время ответа на запросы разработанным сервером в 1.05 – 1.54 раза больше. Это может быть связано с тем, что nginx – более оптимизированное программное обеспечение, предназначенное для профессионалов [24], а также оптимизацией сторонних процессов, таких как логирование и мониторинг дочерних процессов.

Однако нагрузочное тестирование показало, что сервер работает стабильно: программа корректно обработала все запросы, посылаемые ей в рамках исследования.

## ЗАКЛЮЧЕНИЕ

В данной курсовой работе были проанализированы способы проектирования многопользовательских серверов, рассмотрены различные модели, две из которых признаны наиболее эффективными: pre-threading и pre-forking. Были рассмотрены основы сетевого стека Linux.

Для решения поставленной задачи представлены схемы алгоритмов. В соответствии со спроектированными алгоритмами был разработан статический сервер с использованием epoll на базе архитектуры pre-fork.

Решены следующие задачи:

- проанализированы способы проектирования многопользовательских серверов, изучены предоставляемые Unix-системами API для их создания;
- спроектирован статический сервер на основе архитектуры pre-fork с использованием epoll;
- реализован сервер и протестировано разработанное программное обеспечение;
- проведено сравнение результатов нагрузочного тестирования разработанного сервера с nginx.

Таким образом, цель данной курсовой работы выполнена: разработан классический статический сервер для отдачи контента с диска на основе технологий pre-fork и epoll.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Massé, M. REST API Design Cookbook / M. Massé. – Sebastopol: O'Reilly, 2012. – 114 p.
2. Tiwari, L., Pandey, V.G. A Secure Pre-threaded and Pre-forked Unix Client-Server Design for Efficient Handling of Multiple Clients. – 2012. – 5 p.
3. Comparing and Evaluating epoll, select, and poll Event Mechanisms / L. Gammo [et al.] // Proceedings of the 6th Annual Ottawa Linux Symposium. – 2004. – P. 215-225.
4. Стивенс, Р. Раго, С. UNIX. Профессиональное программирование, 2-е издание. – СПб.: Символ-Плюс, 2007. – 1040 с.
5. socket(2) – Linux manual page. Режим доступа: <https://man7.org/linux/man-pages/man2/socket.2.html> (дата обращения: 01.11.2023)
6. bind(2) – Linux manual page. Режим доступа: <https://man7.org/linux/man-pages/man2/bind.2.html> (дата обращения: 01.11.2023)
7. connect(2) – Linux manual page. Режим доступа: <https://man7.org/linux/man-pages/man2/connect.2.html> (дата обращения: 01.11.2023)
8. accept(2) – Linux manual page. Режим доступа: <https://man7.org/linux/man-pages/man2/accept.2.html> (дата обращения: 01.11.2023)
9. fork(2) – Linux manual page. Режим доступа: <https://man7.org/linux/man-pages/man2/fork.2.html> (дата обращения: 01.11.2023)
10. select(2) – Linux manual page. Режим доступа: <https://man7.org/linux/man-pages/man2/select.2.html> (дата обращения: 01.11.2023)
11. poll(2) – Linux manual page. Режим доступа: <https://man7.org/linux/man-pages/man2/poll.2.html> (дата обращения: 01.11.2023)

12. `epoll(2)` – Linux manual page. Режим доступа: <https://man7.org/linux/man-pages/man7/epoll.7.html> (дата обращения: 01.11.2023)
13. `pselect(3p)` – Linux manual page. Режим доступа: <https://man7.org/linux/man-pages/man3/pselect.3p.html> (дата обращения: 01.11.2023)
14. `epoll_ctl(2)` – Linux manual page. Режим доступа: [https://man7.org/linux/man-pages/man2/epoll\\_ctl.2.html](https://man7.org/linux/man-pages/man2/epoll_ctl.2.html) (дата обращения: 01.11.2023)
15. `epoll_ctl(2)` – Linux manual page. Режим доступа: [https://man7.org/linux/man-pages/man2/epoll\\_wait.2.html](https://man7.org/linux/man-pages/man2/epoll_wait.2.html) (дата обращения: 01.11.2023)
16. ISO/IEC 9899:1999(E) — Programming Languages – C. Режим доступа: [https://www.dii.uchile.cl/daespino/files/Iso\\_C\\_1999\\_definition.pdf](https://www.dii.uchile.cl/daespino/files/Iso_C_1999_definition.pdf) (дата обращения: 01.11.2023)
17. GCC, the GNU Compiler Collection. Режим доступа: <https://gcc.gnu.org/> (дата обращения: 01.11.2023)
18. Visual Studio Code. Режим доступа: <https://code.visualstudio.com/> (дата обращения: 01.11.2023)
19. Docker Docs: How to build, share and run applications. Режим доступа: <https://docs.docker.com/> (дата обращения: 15.09.2023)
20. Windows. Режим доступа: <https://www.microsoft.com/ru-ru/windows> (дата обращения: 15.09.2023)
21. Процессор Intel® Core™ i5-8265U. Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/149088/intel-core-i58265u-processor-6m-cache-up-to-3-90-ghz.html> (дата обращения: 15.09.2023)
22. `ab` - Apache HTTP server benchmarking tool. Режим доступа: <https://httpd.apache.org/docs/2.4/programs/ab.html> (дата обращения: 01.11.2023)

23. nginx. Режим доступа: <https://nginx.org/ru/> (дата обращения: 01.11.2023)
24. Tuning NGNIX for Performance. Режим доступа: <https://www.nginx.com/blog/tuning-nginx/> (дата обращения: 01.11.2023)