

Содержание

Введение.....	4
1. Аналитический раздел.....	5
1.1 Постановка задачи.....	5
1.2 Многопоточность в Java Virtual Machine (JVM).....	5
1.3 Анализ требований к синхронизации в модели «читатели-писатели».....	6
1.4 Методы синхронизации потоков в JVM.....	7
1.5 Выводы.....	11
2. Конструкторский раздел.....	12
2.1 Последовательность действий клиент-серверного взаимодействия	12
2.2 Архитектура клиент-серверного взаимодействия.....	13
2.3 Алгоритм работы клиента.....	13
2.4 Алгоритм работы сервера.....	13
2.5 Блок-схема алгоритма работы клиента.....	14
2.6 Блок-схема алгоритма работы сервера.....	15
2.7 Используемые структуры данных.....	16
2.8 Взаимодействие через подсистему ввода/вывода.....	16
3. Технологический раздел.....	17
3.1 Выбор языка программирования.....	17
3.2 Выбор среды разработки	18
3.3 Особенности реализации.....	18
4. Исследовательский раздел.....	23
4.1 Блокировка и запись.....	23
4.2 Чтение.....	24
Заключение	25
Список используемых источников.....	27
Приложение А (Исходный код клиент-серверного приложения).....	28

Введение

В современных операционных системах (ОС) эффективное управление совместным доступом к общим ресурсам является ключевой задачей. В условиях многозадачности и многопоточности процессы конкурируют за ограниченные ресурсы, и синхронизация их доступа становится критически важной для обеспечения корректности и производительности системы. Основным механизмом, обеспечивающим безопасный доступ к разделяемым ресурсам, является механизм синхронизации, позволяющий предотвращать взаимные блокировки и обеспечивать целостность данных.

Одной из типовых задач синхронизации является задача «читатели-писатели». Суть данной задачи заключается в разделении доступа к общему ресурсу между процессами-читателями и процессами-писателями: несколько процессов могут одновременно читать данные, но операции записи требуют эксклюзивного доступа. В классическом подходе, если ресурс занят «писателем», доступ для чтения и записи блокируется для всех других процессов. В данной работе рассматривается модифицированная версия этой модели, в которой обеспечивается возможность параллельного множественного чтения данными «читателями», при условии отсутствия активных «писателей».

1. Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу требуется разработать и реализовать модель клиент-серверного взаимодействия по принципу «читатели-писатели». Основная цель заключается в создании системы, в которой несколько клиентов могут одновременно читать данные, и только один клиент может писать. Это позволит избежать конфликтов доступа и поддерживать согласованность данных.

Такой подход особенно важен для приложений, где конкурентный доступ к данным требует обеспечения целостности, минимизации времени ожидания и оптимизации работы системы при большом количестве запросов на чтение и запись. Задача также включает анализ методов синхронизации и многопоточности, реализованных в Java Virtual Machine (JVM).

Для достижения поставленной цели необходимо:

- Проанализировать основные методы реализации модели «читатели-писатели» в многопоточной среде (JVM).
- Определить подходящие методы синхронизации, которые позволят обеспечить конкурентный доступ к ресурсу.
- Разработать алгоритм клиент-серверного взаимодействия с использованием инструментов Java для многопоточности и синхронизации.

1.2 Многопоточность в Java Virtual Machine (JVM)

Так как реализация курсового проекта осуществляется с использованием языка программирования Java, важно рассмотреть особенности работы с потоками в Java Virtual Machine (JVM). JVM обеспечивает платформу-независимость, что позволяет одинаково эффективно управлять потоками как на Windows, так и на Linux и других операционных системах. Основными преимуществами JVM являются структурированный подход к многопоточности и высокая абстракция, которая облегчает разработку многопоточных приложений.

Многопоточность в JVM поддерживается на уровне библиотеки `java.util.concurrent`, которая предоставляет множество классов и интерфейсов для работы с потоками и синхронизацией. Среди них стоит выделить `Thread`, `Runnable`, инструменты синхронизации, такие как `ReentrantLock`, `Semaphore`, и `ReentrantReadWriteLock`.

```

public class SimpleThreadExample {
    public static void main(String[] args) {
        Runnable task = () -> {
            System.out.println("Выполняется поток: " +
Thread.currentThread().getName());
        };
        Thread thread = new Thread(task);
        thread.start();
    }
}

```

Листинг 1. Реализация потоков с использованием класса Thread и Runnable

1.3 Анализ требований к синхронизации в модели «читатели-писатели»

Модель «читатели-писатели» требует механизмов синхронизации, которые позволяют безопасно организовать конкурентный доступ к ресурсу следующим образом:

- Параллельное чтение: несколько клиентов могут одновременно читать данные, если нет активной записи.
- Эксклюзивная запись: запись возможна только при условии, что ни один другой клиент не читает или не записывает данные.

Для этой цели выбран механизм ReentrantReadWriteLock, который предоставляет две блокировки:

- Чтение (readLock): позволяет множественным потокам одновременно читать данные ресурса, если в текущий момент никто не выполняет операцию записи.
- Запись (writeLock): обеспечивает эксклюзивный доступ к ресурсу, блокируя все другие операции на время выполнения записи.

Использование ReentrantReadWriteLock позволяет достигать баланса между безопасностью данных и производительностью в условиях высокой конкурентности между операциями чтения и записи.

```

import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadWriteLockExample {
    private int sharedResource = 0;
    private final ReentrantReadWriteLock lock = new
ReentrantReadWriteLock();

    // Метод для чтения ресурса
    public void read() {
        lock.readLock().lock();
        try {
            System.out.println(Thread.currentThread().getName() + "
читает ресурс: " + sharedResource);
        } finally {
            lock.readLock().unlock();
        }
    }

    // Метод для записи ресурса
    public void write(int value) {
        lock.writeLock().lock();
        try {
            sharedResource = value;
            System.out.println(Thread.currentThread().getName() + "
записывает значение: " + sharedResource);
        } finally {
            lock.writeLock().unlock();
        }
    }
}

```

Листинг 2. Использование ReentrantReadWriteLock для синхронизации доступа к общему ресурсу

1.4. Методы синхронизации потоков в JVM

Рассмотрим основные механизмы синхронизации потоков, предоставляемые JVM, а также их применимость для задачи «читатели-писатели». В ходе анализа различных методов был сделан выбор в пользу использования ReentrantReadWriteLock благодаря его реализации: параллельный доступ для операций чтения и эксклюзивный доступ для операций записи, что позволяет сбалансировать производительность и безопасность данных.

1. Мониторы (synchronized):

Встроенный в язык Java механизм, позволяющий синхронизировать доступ к методам или блокам кода.

- Плюсы:

Простота использования и интеграция на уровне языка. У каждого объекта, создаваемого в Java, уже реализован монитор по умолчанию.

- Минусы:

Мониторы не позволяют разделить операции чтения и записи, что ограничивает параллелизм. Это может приводить к снижению производительности при частых операциях чтения, так как блокируется весь метод или блок кода.

```
public class SynchronizedExample {
    private int sharedResource = 0;

    public synchronized void read() {
        System.out.println(Thread.currentThread().getName() + "
читает ресурс: " + sharedResource);
    }

    public synchronized void write(int value) {
        sharedResource = value;
        System.out.println(Thread.currentThread().getName() + "
записывает значение: " + sharedResource);
    }
}
```

Листинг 3. Мониторы (synchronized)

2. Мьютексы (ReentrantLock):

Предоставляет возможность рекурсивного захвата блокировки и более явное управление блокировкой.

- Плюсы:

Поддержка справедливости (fairness) и возможность более точного контроля. Поток, ожидающий на блокировке дольше всех, будет обслужен первым.

- Минусы:

Не позволяет разделить доступ на чтение и запись, что ограничивает параллелизм в условиях частых операций чтения.

```
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockExample {
    private int sharedResource = 0;
    private final ReentrantLock lock = new ReentrantLock();

    public void read() {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "
читает ресурс: " + sharedResource);
        } finally {
            lock.unlock();
        }
    }

    public void write(int value) {
        lock.lock();
        try {
            sharedResource = value;
            System.out.println(Thread.currentThread().getName() + "
записывает значение: " + sharedResource);
        } finally {
            lock.unlock();
        }
    }
}
```

Листинг 4. Мьютексы (ReentrantLock)

3. Семафоры (Semaphore):

Используются для ограничения количества потоков, имеющих доступ к ресурсу.

- Плюсы:

Возможность использования для управления пулом ресурсов.

- Минусы:

Сложность в управлении при разделении доступа на чтение и запись. Необходимо вручную управлять количеством разрешений для корректного поведения.

```
import java.util.concurrent.Semaphore;

public class SemaphoreExample {
    private final Semaphore semaphore = new Semaphore(3); //
    Ограничиваем доступ тремя ресурсами

    public void accessResource() {
        try {
            semaphore.acquire(); // Пытаемся захватить ресурс
            System.out.println(Thread.currentThread().getName()
+ " получил доступ к ресурсу");
            Thread.sleep(2000); // Имитация работы с ресурсом
            System.out.println(Thread.currentThread().getName()
+ " освободил ресурс");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            semaphore.release(); // Освобождаем ресурс
        }
    }

    public static void main(String[] args) {
        SemaphoreExample example = new SemaphoreExample();
        for (int i = 0; i < 5; i++) {
            new Thread(example::accessResource, "Поток " + (i +
1)).start();
        }
    }
}
```

Листинг 5. Семафоры (Semaphore)

1.5. Выводы

Для реализации клиент-серверного приложения на основе задачи «читатели-писатели» были рассмотрены различные механизмы синхронизации потоков, доступные в Java Virtual Machine. В процессе анализа особенностей многопоточности, таких как поддержка конкурентного доступа к общим ресурсам, оценивались методы, включая мониторы (synchronized), мьютексы (ReentrantLock), семафоры и ReentrantReadWriteLock. Каждый из них имеет свои особенности, преимущества и недостатки, которые могут влиять на производительность и целостность данных в многопользовательских системах.

Наиболее подходящим методом для решения задачи «читатели-писатели» оказался механизм ReentrantReadWriteLock, обеспечивающий разделение доступа на чтение и запись. Его ключевые преимущества включают возможность параллельного чтения несколькими потоками и эксклюзивный доступ для записи, что снижает конкуренцию за ресурс и минимизирует время ожидания операций, особенно когда операций чтения значительно больше, чем операций записи. Это делает ReentrantReadWriteLock особенно эффективным в сценариях, где необходимо обеспечить максимальную производительность для чтения при сохранении безопасности данных для операций записи.

2. Конструкторский раздел

2.1 Последовательность действий клиент-серверного взаимодействия

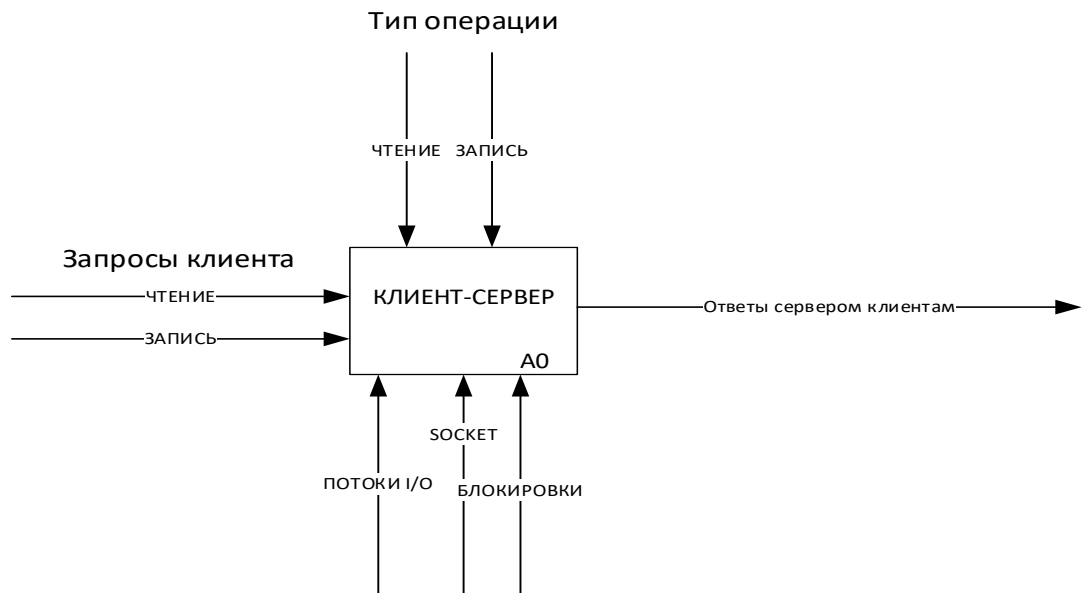


Рисунок 1. Последовательность действий работы приложения

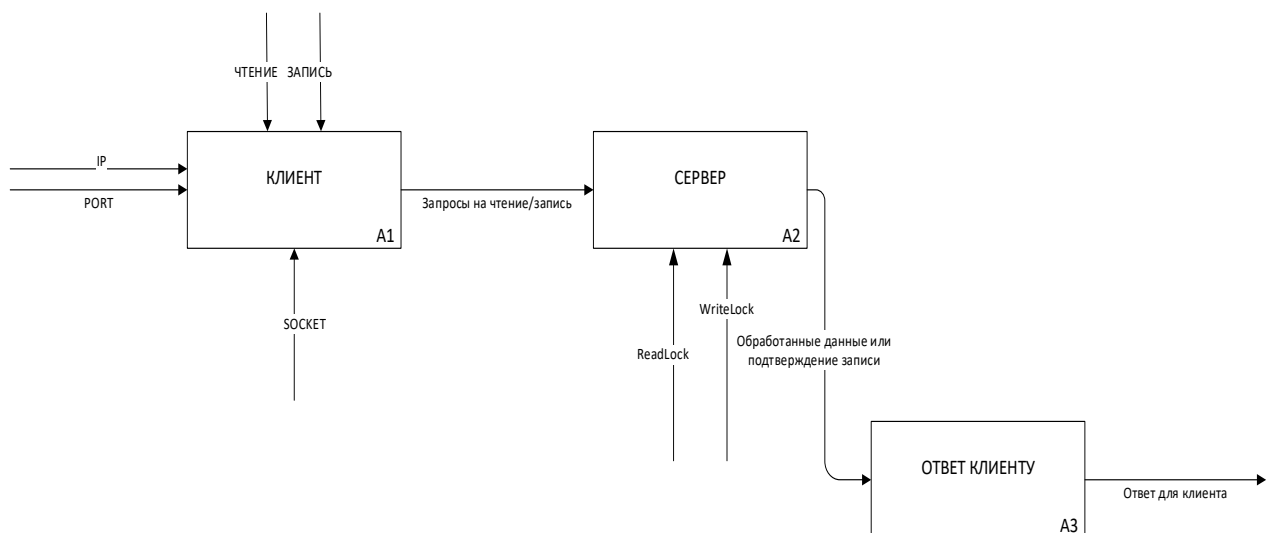


Рисунок 2. Последовательность действий работы приложения

2.2 Архитектура клиент-серверного взаимодействия

Клиент подключается к серверу и выполняет либо чтение, либо запись в зависимости от случайного выбора (см. Рис.1, Рис. 2).

Сервер обрабатывает запросы, управляет доступом к общим ресурсам, обеспечивая многопользовательский доступ (см. Рис.1, Рис. 2).

2.3 Алгоритм работы клиента

— Подключение к серверу:

Клиент инициирует соединение с сервером по определенному IP-адресу и порту.

— Определение типа операции:

С использованием генератора случайных чисел определяется, будет ли клиент выполнять чтение или запись.

— Выполнение операции:

Если операция чтения — клиент отправляет запрос "read" и получает ответ от сервера.

Если операция записи — клиент генерирует данные, отправляет запрос "write <данные>" и ожидает подтверждения от сервера.

— Завершение работы:

Закрывание соединения и освобождение ресурсов.

2.4 Алгоритм работы сервера

— Принятие подключения:

Сервер слушает порт и принимает подключения от клиентов.

— Создание потока для клиента:

Для каждого клиента создается отдельный поток для асинхронной обработки запросов.

— Обработка запроса:

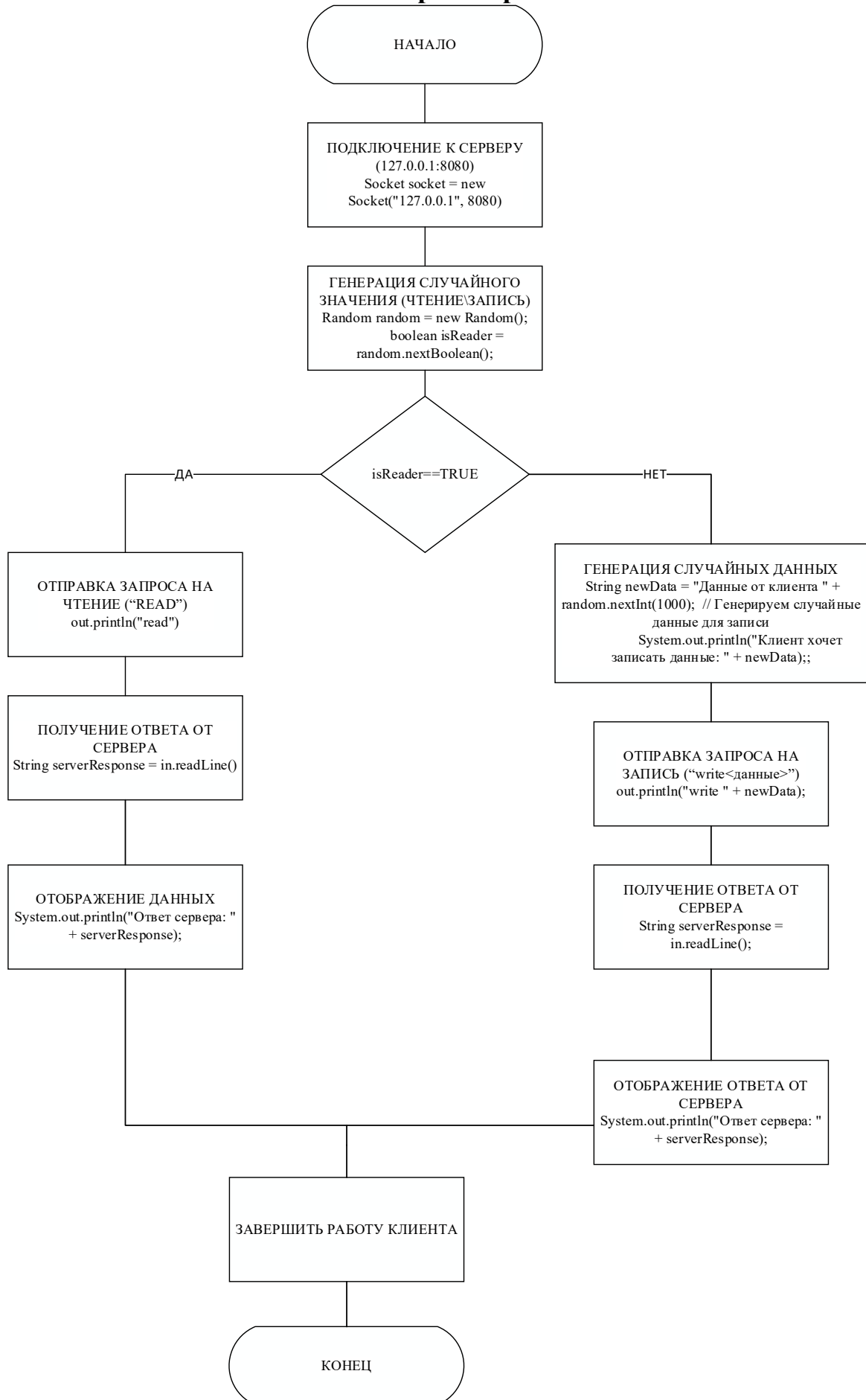
- Чтение: при получении запроса на чтение захватывается блокировка чтения (readLock), после чего данные отправляются клиенту.

- Запись: при получении запроса на запись сервер пытается захватить блокировку записи (writeLock), чтобы обеспечить эксклюзивный доступ к ресурсу.

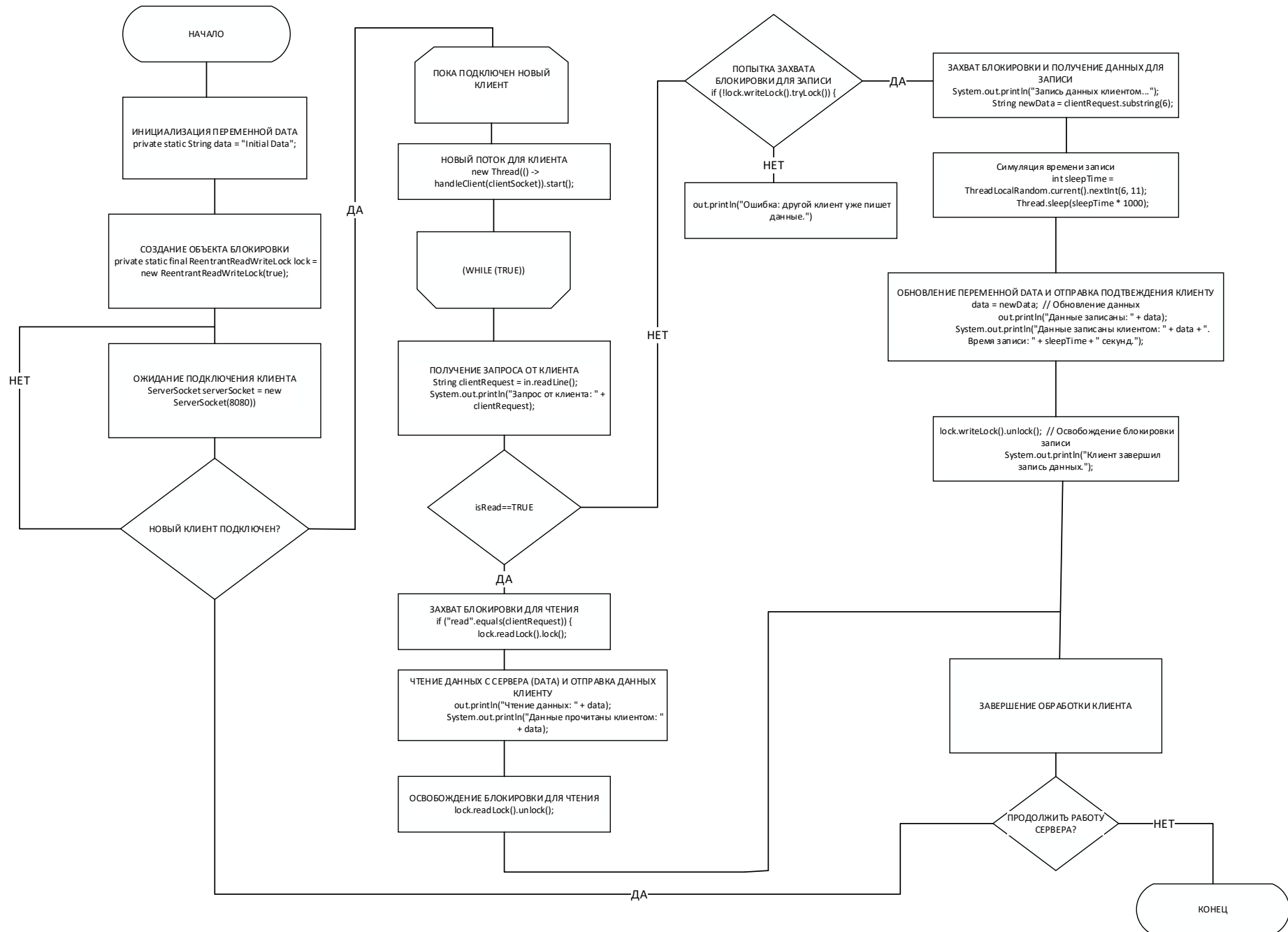
— Завершение работы:

Поток клиента завершается, сокет закрывается.

2.5 Блок-схема алгоритма работы клиента



2.6 Блок-схема алгоритма работы сервера



2.7 Используемые структуры данных

Для реализации клиент-серверной модели были выбраны следующие структуры данных:

- Класс Client: отвечает за подключение и взаимодействие с сервером.
- Методы:
 - connect(): подключение к серверу.
 - read(): чтение данных с сервера.
 - write(): запись данных на сервер.
- Класс Server: обрабатывает запросы клиентов и управляет доступом к общим ресурсам.
- Методы:
 - accept(): принимает подключение клиента.
 - handleClientRequest(): обрабатывает запросы на чтение или запись.
 - ReentrantReadWriteLock: служит для синхронизации операций чтения и записи. Он предоставляет две блокировки:
 - readLock: разрешает параллельное чтение нескольким потокам.
 - writeLock: предоставляет эксклюзивный доступ к ресурсу на время записи.

2.8 Взаимодействие через подсистему ввода/вывода

Взаимодействие между клиентом и сервером реализовано через стандартные потоки ввода/вывода с использованием следующих компонентов:

BufferedReader и PrintWriter обеспечивают чтение и запись данных между клиентом и сервером.

Каждый клиент, подключающийся к серверу, получает уникальные потоки, что позволяет обеспечивать независимость операций и безопасность данных.

3. Технологический раздел

3.1 Выбор языка программирования

Для разработки данного клиент-серверного приложения был выбран язык программирования Java, так как он предоставляет следующие возможности для реализации многопоточных и сетевых приложений.

Основные причины выбора Java следующие:

- Платформенная независимость: Приложения, написанные на Java, исполняются в виртуальной машине Java (JVM), что обеспечивает платформенную независимость. Это позволяет использовать одно и то же приложение на различных платформах (Windows, Linux, macOS), что особенно важно в случае клиент-серверных решений, где сервер и клиенты могут функционировать в разных средах.
- Поддержка многопоточности: Java предоставляет встроенные инструменты для работы с многопоточностью, такие как классы Thread и библиотеки из java.util.concurrent. Это позволяет разрабатывать приложения, способные обрабатывать несколько запросов одновременно, что критично для серверов, работающих с большим количеством подключений. В данной реализации используются Thread и ReentrantReadWriteLock для поддержки эффективного выполнения задач.
- Библиотека стандартных классов: Стандартные библиотеки Java (java.net, java.io, java.util.concurrent) предоставляют множество классов и интерфейсов для работы с сетевыми взаимодействиями, потоками и синхронизацией. Это упрощает разработку сетевых приложений, снижая необходимость создания низкоуровневых решений.
- Безопасность: Встроенные механизмы безопасности Java, такие как управление памятью и автоматическая сборка мусора, значительно снижают вероятность ошибок, связанных с утечкой памяти, что особенно важно при разработке сетевых многопользовательских приложений.

3.2 Выбор среды разработки

Операционная система: Linux Ubuntu 24.04.1 LTS

Редактор кода: GNU Nano, версия 7.2

Интерпретатор (JVM): OpenJDK Runtime Environment (build 21.0.4)

Компилятор: javac 21.0.4

3.3 Особенности реализации

— Использование сокетов:

Для реализации сетевого взаимодействия между клиентом и сервером используются сокет (Socket и ServerSocket). Это позволяет клиентам устанавливать соединение с сервером и обмениваться данными.

Серверный сокет (ServerSocket serverSocket = new ServerSocket(8080)) инициализируется на порту 8080 и начинает прослушивание входящих подключений. При каждом подключении создается отдельный клиентский сокет для взаимодействия с данным клиентом.

Клиентский сокет (Socket socket = new Socket("127.0.0.1", 8080)) используется для подключения клиента к серверу. После установления соединения клиент может отправлять запросы серверу и получать от него ответы.

```
try (ServerSocket serverSocket = new ServerSocket(8080)) {
    System.out.println("Сервер запущен и ожидает подключения...");
    while (true) {
        Socket clientSocket = serverSocket.accept();
        System.out.println("Клиент подключен: " +
clientSocket.getInetAddress());
        new Thread(() -> handleClient(clientSocket)).start();
    }
} catch (IOException e) {
    System.err.println("Ошибка при запуске сервера: " +
e.getMessage());
    e.printStackTrace();
}
```

Листинг 6. Реализация сокетов на стороне сервера

— Многопоточность:

Для обеспечения параллельной обработки нескольких подключений сервер создает отдельный поток для каждого клиента. Это позволяет обрабатывать несколько запросов одновременно, что повышает общую производительность и отзывчивость приложения

```
new Thread(() -> handleClient(clientSocket)).start();
```

Листинг 7. Создание потока

Данный подход реализован с использованием лямбда-функций, которые запускаются в новом потоке. Такой способ минимизирует блокировку при поступлении множества клиентских подключений и гарантирует эффективное использование процессорного времени.

```

private void handleClient(Socket clientSocket) {
    try (BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true)) {

        String clientRequest;
        while ((clientRequest = in.readLine()) != null) {
            System.out.println("Получен запрос: " + clientRequest);
            if ("read".equals(clientRequest)) {
                lock.readLock().lock();
                try {
                    out.println("Чтение данных: " + sharedResource);
                } finally {
                    lock.readLock().unlock();
                }
            } else if (clientRequest.startsWith("write")) {
                if (!lock.writeLock().tryLock()) {
                    out.println("Ошибка: другой клиент уже пишет
данные.");
                } else {
                    try {
                        String newData = clientRequest.substring(6);
                        sharedResource = newData;
                        out.println("Запись выполнена успешно.");
                    } finally {
                        lock.writeLock().unlock();
                    }
                }
            } else {
                out.println("Неизвестный запрос");
            }
        }
    } catch (IOException e) {
        System.err.println("Ошибка при обработке клиента: " +
e.getMessage());
        e.printStackTrace();
    }
}

```

Листинг 8. Обработки клиента в отдельном потоке

— Синхронизация доступа к общим ресурсам:

Для предотвращения конфликтов при одновременном доступе к общим ресурсам используется `ReentrantReadWriteLock`.

Чтение данных выполняется с использованием `readLock()`, который позволяет нескольким клиентам одновременно читать данные, если нет операций записи.

Запись данных выполняется через `writeLock()`, который предоставляет эксклюзивный доступ к ресурсу, предотвращая чтение или запись другими потоками в это время. Такой механизм гарантирует консистентность данных и предотвращает конфликты между потоками за общие ресурсы.

```
private static final ReentrantReadWriteLock lock = new
ReentrantReadWriteLock(true);

public void read() {
    lock.readLock().lock();
    try {
        System.out.println(Thread.currentThread().getName() + " читает
ресурс: " + sharedResource);
    } finally {
        lock.readLock().unlock();
    }
}

public void write(int value) {
    lock.writeLock().lock();
    try {
        sharedResource = value;
        System.out.println(Thread.currentThread().getName() + "
записывает значение: " + sharedResource);
    } finally {
        lock.writeLock().unlock();
    }
}
```

Листинг 9. Использование ReentrantReadWriteLock

— Обработка ошибок

В коде реализована обработка исключений (`IOException`), что позволяет избежать сбоев при работе с сетью и обеспечить устойчивость приложения.

```
catch (IOException e) {
    System.err.println("Ошибка подключения: " + e.getMessage());
    e.printStackTrace();
}
```

Листинг 10. Обработка ошибок

Это решение позволяет своевременно выявлять и исправлять ошибки, связанные с подключением и передачей данных, а также выводит сообщения об ошибках в стек вызовов, что помогает при отладке и анализе проблем.

— Обработка запросов клиентов:

Сервер обрабатывает запросы клиентов, разделяя их на операции чтения и записи:

```
if ("read".equals(clientRequest)) {
    lock.readLock().lock();
    try {
        out.println("Чтение данных: " + data);
    } finally {
        lock.readLock().unlock();
    }
}
```

Листинг 11. Обработка чтения

Если запрос клиента — это чтение, сервер захватывает блокировку чтения, отправляет данные клиенту и освобождает блокировку после завершения операции.

```
else if (clientRequest.startsWith("write")) {
    if (!lock.writeLock().tryLock()) {
        out.println("Ошибка: другой клиент уже пишет данные.");
    } else {
        String newData = clientRequest.substring(6);
        data = newData; // Обновление данных
    }
}
```

Листинг 12. Обработка записи

Если запрос клиента — это запись, сервер проверяет доступность блокировки записи и обновляет данные, если она доступна. В противном случае клиент получает уведомление об ошибке.

— Генерация случайных данных

Клиенты генерируют случайные данные для записи, что делает приложение более динамичным и позволяет тестировать его в различных сценариях:

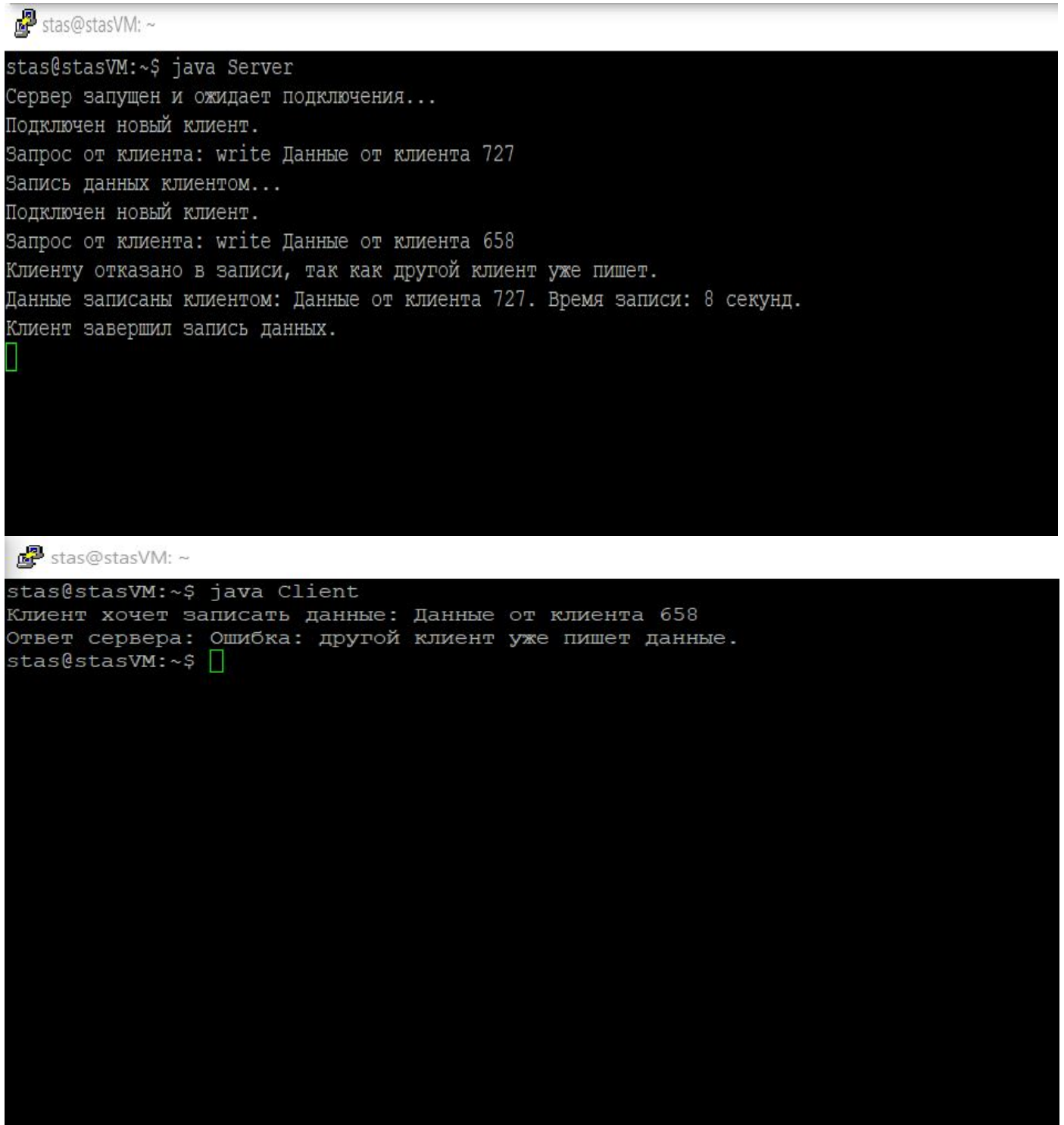
```
String newData = "Данные от клиента " + random.nextInt(1000);
```

Листинг 13. Генерация случайных данных

Эта особенность позволяет проверять, как сервер обрабатывает запросы и управляет состоянием данных.

4. Исследовательский раздел

4.1 Блокировка и запись



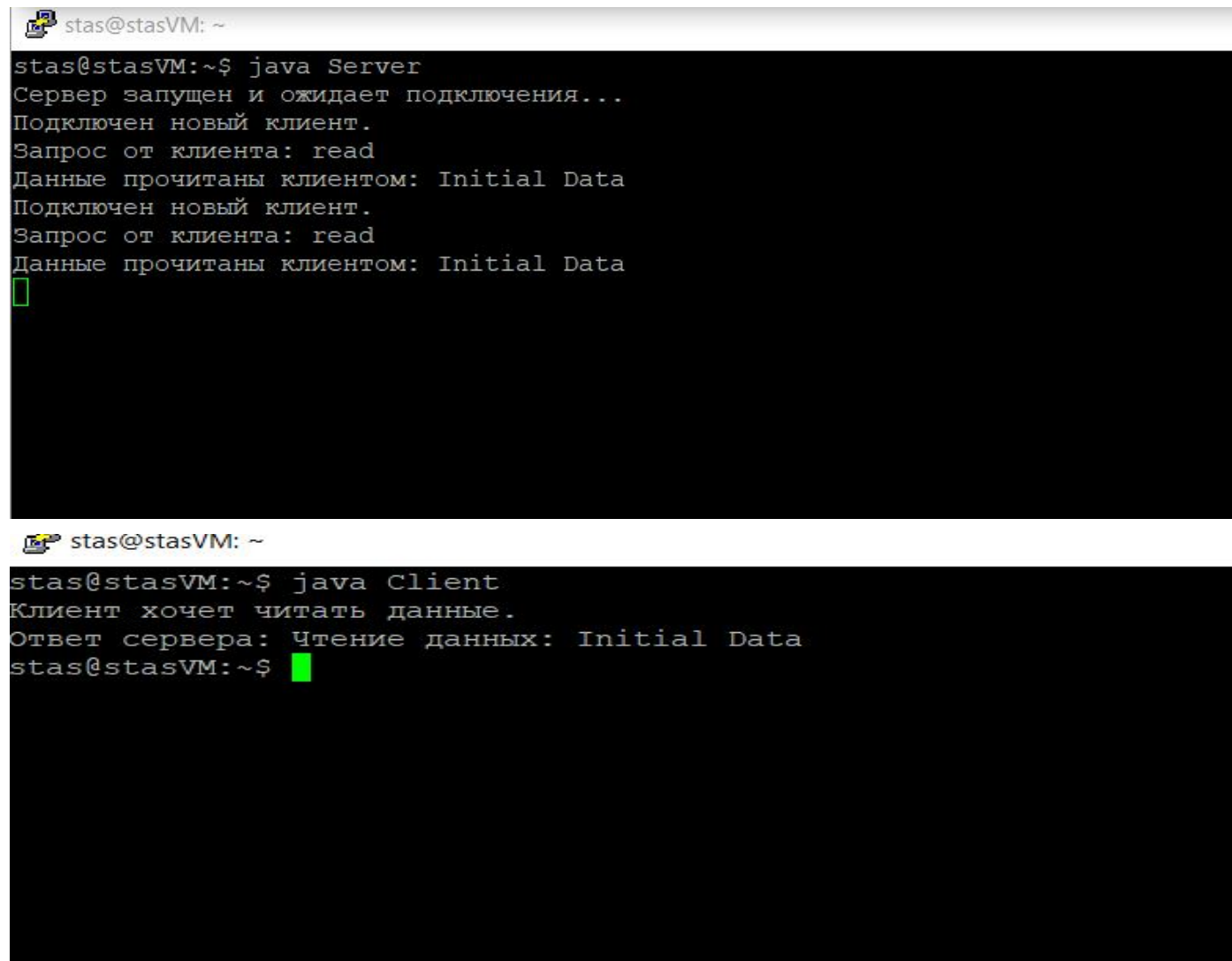
```
stas@stasVM: ~  
stas@stasVM:~$ java Server  
Сервер запущен и ожидает подключения...  
Подключен новый клиент.  
Запрос от клиента: write Данные от клиента 727  
Запись данных клиентом...  
Подключен новый клиент.  
Запрос от клиента: write Данные от клиента 658  
Клиенту отказано в записи, так как другой клиент уже пишет.  
Данные записаны клиентом: Данные от клиента 727. Время записи: 8 секунд.  
Клиент завершил запись данных.  
[  
  
stas@stasVM: ~  
stas@stasVM:~$ java Client  
Клиент хочет записать данные: Данные от клиента 658  
Ответ сервера: Ошибка: другой клиент уже пишет данные.  
stas@stasVM:~$ [
```

Рисунок 3. Работа клиента и сервера

Сервер обрабатывает несколько клиентских соединений и использует блокировку `writeLock` для синхронизации доступа к общему ресурсу (`data`). Когда клиент отправляет запрос на запись, сервер проверяет наличие активной блокировки.

Если блокировка уже захвачена другим клиентом, текущий клиент получает отказ, что помогает избежать конфликтов и сохранить целостность данных. Таким образом, одновременно несколько клиентов не могут выполнять запись.

4.2 Чтение



```
stas@stasVM: ~  
stas@stasVM:~$ java Server  
Сервер запущен и ожидает подключения...  
Подключен новый клиент.  
Запрос от клиента: read  
Данные прочитаны клиентом: Initial Data  
Подключен новый клиент.  
Запрос от клиента: read  
Данные прочитаны клиентом: Initial Data  
█  
  
stas@stasVM: ~  
stas@stasVM:~$ java Client  
Клиент хочет читать данные.  
Ответ сервера: Чтение данных: Initial Data  
stas@stasVM:~$ █
```

Рисунок 4. Работа клиента и сервера

Клиент отправляет запрос на чтение с помощью команды "read", сервер захватывает блокировку чтения (readLock). Это позволяет нескольким клиентам одновременно читать данные, если запись в текущий момент не осуществляется. Сервер успешно обрабатывает запрос, отправляет данные клиенту и освобождает блокировку чтения. Такой подход обеспечивает параллельное чтение несколькими клиентами, что улучшает производительность при большом количестве запросов на чтение.

Заключение

В ходе выполнения курсовой работы была разработана и реализована модель клиент-серверного взаимодействия по принципу «читатели-писатели» с использованием средств многопоточности и синхронизации в Java. Основные направления работы и достигнутые результаты можно представить следующим образом:

- Проанализированы основные методы реализации многопоточной модели «читатели-писатели» в JVM, включая использование таких механизмов, как мониторы (`synchronized`), мьютексы (`ReentrantLock`), семафоры и `ReentrantReadWriteLock`.
- Рассмотрены и выбраны подходящие методы синхронизации для обеспечения безопасного и эффективного доступа к общим ресурсам. В качестве основного механизма был выбран `ReentrantReadWriteLock`, который позволяет реализовать параллельное чтение и эксклюзивную запись.
- Разработана архитектура клиент-серверного приложения, включающая многопоточный сервер и клиентов, работающих с общим ресурсом. Клиенты взаимодействуют с сервером посредством сокетов, что обеспечивает гибкость и масштабируемость приложения.
- Предложены и реализованы алгоритмы взаимодействия клиента и сервера, обеспечивающие конкурентный доступ к данным. Были разработаны алгоритмы обработки запросов чтения и записи с использованием блокировок `readLock` и `writeLock` для управления доступом к общему ресурсу.
- Разработано консольное клиент-серверное приложение на языке программирования Java, которое продемонстрировало успешное параллельное взаимодействие нескольких клиентов с общим ресурсом на сервере.
- Тестирование показало, что разработанное приложение эффективно управляет доступом к ресурсу, минимизируя конкуренцию и время ожидания для операций чтения, что особенно важно в условиях большого количества клиентских запросов на чтение и запись.

Таким образом, реализация предложенного подхода позволила обеспечить безопасный и оптимальный доступ к общим ресурсам в многопользовательской среде. Применение ReentrantReadWriteLock позволило достигнуть максимальной производительности операций чтения при сохранении безопасности данных для операций записи, что делает предложенное решение особенно актуальным для систем, где доля операций чтения значительно превышает количество операций записи.

Список используемых источников

1. Душутина Е.В. Межпроцессные взаимодействия в операционных системах. Учебное пособие – СПб.: 2014 – 135 стр.
2. Огинский А.А., Набатчиков А.М., Бурлак Е.А. Организация межпотокowego взаимодействия с использованием объектов ядра операционной системы – С. 52-56. Вестник компьютерных и информационных технологий. №8. Август 2012.
3. Щупак Ю. А. Win32 API. Разработка приложений для Windows – СПб.: Питер, 2008 – 592 стр.
4. Э. Таненбаум. Современные операционные системы – СПб.:Питер, 2010. – 1120 стр.
5. Герберт Шилдт. Полное руководство Java. 12-е издание. 2022 год. – 1344 стр.
6. Гетц Брайан, Пайерлс Тим, Блох Джошуа, Боубер Джозеф, Холмс Дэвид, Ли Даг. Java Concurrency на практике-СПб.:Питер, 2022 г.- 464 стр.
7. Таненбаум Э., Бос Х. Современные операционные системы / Таненбаум Э., Бос Х. ; пер. с англ.Леонтьева А., Малышева М., Вильчинский Н. - 4-е изд. - СПб. : Питер, 2020. - 1119 с. : рис., табл. (Классика computer science). - Библиогр.: с. 1010-1119. - ISBN 978-5-4461-1155-8.
8. Крищенко В. А., Рязанова Н. Ю. Сервисы Windows : учеб. пособие / Крищенко В. А., Рязанова Н. Ю. ; МГТУ им. Н. Э. Баумана. - М. : Изд-во МГТУ им. Н. Э. Баумана, 2011. - 47 с. - Библиогр.: с. 47.
9. Крищенко, В. А. Основы программирования в ядре операционной системы GNU/Linux : учебное пособие / В. А. Крищенко, Н. Ю. Рязанова. — Москва : МГТУ им. Н.Э. Баумана, 2010. — 34 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: <https://e.lanbook.com/book/58435>
10. Крищенко В. А., Рязанова Н. Ю. Основы программирования в ядре операционной системы GNU/LINUX : учеб. пособие / Крищенко В. А., Рязанова Н. Ю. ; МГТУ им. Н. Э. Баумана. - М. : Изд-во МГТУ им. Н. Э. Баумана, 2010. - 34 с. - Библиогр.: с. 34.
11. Гостев, И. М. Операционные системы : учебник и практикум для вузов / И. М. Гостев. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2024. — 164 с. — (Высшее образование). — ISBN 978-5-534-04520-8.
12. Гостев, И. М. Операционные системы : учебник и практикум для вузов / И. М. Гостев. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2024. — 164 с. — (Высшее образование). — ISBN 978-5-534-04520-8.

ПРИЛОЖЕНИЕ А

(Исходный код клиент-серверного приложения)

```
import java.io.*;
import java.net.*;
import java.util.concurrent.locks.ReentrantReadWriteLock;
import java.util.concurrent.ThreadLocalRandom;

public class Server {

    private static String data = "Initial Data"; // Общая переменная
    для хранения данных
    private static final ReentrantReadWriteLock lock = new
    ReentrantReadWriteLock(true); // Блокировка для синхронизации
    доступа

    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(8080)) {
            System.out.println("Сервер запущен и ожидает
подключения...");

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Подключен новый клиент.");
                new Thread(() -> handleClient(clientSocket)).start();
// Создание отдельного потока для каждого клиента
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void handleClient(Socket clientSocket) {
        try (BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true)) {

            String clientRequest = in.readLine();
            System.out.println("Запрос от клиента: " +
clientRequest);
            if ("read".equals(clientRequest)) {
                lock.readLock().lock(); // Захват блокировки для
чтения

                try {
                    out.println("Чтение данных: " + data);
                    System.out.println("Данные прочитаны клиентом: "
+ data);
                }
            }
        }
    }
}
```

```

        } finally {
            lock.readLock().unlock(); // Освобождение
блокировки
        }
    } else if (clientRequest.startsWith("write")) {
        // Проверяем, если блокировка записи уже активна,
возвращаем ошибку
        if (!lock.writeLock().tryLock()) { // tryLock()
немедленно возвращает false, если блокировка уже захвачена
            out.println("Ошибка: другой клиент уже пишет
данные.");
            System.out.println("Клиенту отказано в записи,
так как другой клиент уже пишет.");
        } else {
            try {
                System.out.println("Запись данных
клиентом...");
                String newData = clientRequest.substring(6);

                // Симуляция времени записи
                int sleepTime =
ThreadLocalRandom.current().nextInt(6, 11);
                Thread.sleep(sleepTime * 1000);

                data = newData; // Обновление данных
                out.println("Данные записаны: " + data);
                System.out.println("Данные записаны клиентом:
" + data + ". Время записи: " + sleepTime + " секунд.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.writeLock().unlock(); // Освобождение
блокировки записи
                System.out.println("Клиент завершил запись
данных.");
            }
        }
    } else {
        out.println("Неизвестная команда.");
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        clientSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

```

import java.io.*;
import java.net.*; // библиотеки для работы с сетевыми сокетами
import java.util.Random;

public class Client {

    public static void main(String[] args) {
        // Подключаемся к серверу по адресу localhost (127.0.0.1) и
        // порту 8080
        try (Socket socket = new Socket("127.0.0.1", 8080);
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream())); // Поток для получения
данных от сервера
            PrintWriter out = new
PrintWriter(socket.getOutputStream(), true)) { // Поток для отправки
данных на сервер
            // Генерируем случайное значение, чтобы определить, будет
клиент читать или писать
            Random random = new Random();
            boolean isReader = random.nextBoolean(); // Случайно
выбираем роль клиента
            if (isReader) {
                // Клиент хочет прочитать данные
                System.out.println("Клиент хочет читать данные.");
                out.println("read"); // Отправляем запрос на чтение
данных
                String serverResponse = in.readLine(); // Получаем
ответ от сервера
                System.out.println("Ответ сервера: " +
serverResponse); // Выводим ответ сервера
            } else {
                // Клиент хочет записать данные
                String newData = "Данные от клиента " +
random.nextInt(1000); // Генерируем случайные данные для записи
                System.out.println("Клиент хочет записать данные: " +
newData);
                out.println("write " + newData); // Отправляем
запрос на запись данных
                String serverResponse = in.readLine(); // Получаем
ответ от сервера
                System.out.println("Ответ сервера: " +
serverResponse); // Выводим ответ сервера
            }
            } catch (IOException e) {
                e.printStackTrace(); // Обработка ошибок при подключении
или передаче данных
            }
        }
    }
}

```