МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Ордена трудового Красного Знамени федеральное государственное бюджетное
образовательное учреждение высшего образования
«Московский технический университет связи и информатики»

Кафедра Математическая кибернетика и информационные технологии

Отчет по лабораторной работе №7

Выполнил студент группы:
БПИ2401
Мещеряков Кирилл Владимироврич
Руководитель:
Харрасов Камиль Раисович

Москва, 2025

Цель работы:

Изучение принципов многопоточного программирования в Java, освоение различных механизмов создания и синхронизации потоков. Приобретение практических навыков разработки многопоточных приложений с использованием классов Thread, Runnable, ExecutorService, а также средств синхронизации (synchronized, Lock, Semaphore, CountDownLatch, CyclicBarrier). Получение опыта решения типовых задач параллельного программирования: вычисления суммы элементов массива, поиска максимального элемента в матрице и реализации системы управления ресурсами на примере задачи о складе и грузчиках.

Индивидуальное задание:

Задание 1.

Реализация многопоточной программы для вычисления сум- мы элементов массива.

Вариант 1. Создать два потока, которые будут вычислять сумму элементов массива по половинкам, после чего результаты будут складываться в главном потоке.

Вариант 2. Создать пул потоков с помощью класса Executor- Service и разделить массив на равные части, каждую из которых будет обрабатывать отдельный поток. После завершения работы всех потоков результаты будут складываться в главном потоке.

Задание 2.

Реализация многопоточной программы для поиска наиболь- шего элемента в матрице.

Вариант 1. Создать несколько потоков, каждый̆ из которых будет обрабатывать свою строку матрицы. После завершения работы всех потоков результаты будут сравниваться в главном потоке для нахождения наибольшего элемента.

Вариант 2. Создать пул потоков с помощью класса Executor- Service и разделить матрицу на равные части, каждую из кото- рых будет обрабатывать отдельный̆ поток. После завершения работы всех потоков результаты будут сравниваться в главном потоке для нахождения наибольшего элемента.

Задание 3.

У вас есть склад с товарами, которые нужно перенести на другой̆ склад. У каждого товара есть свой вес. На складе ра- ботают 3 грузчика. Грузчики могут переносить товары одно- временно, но суммарный̆ вес товаров, переносимый̆ ими за одну итерацию, не может превышать 150 кг. Как только грузчики со- берут 150 кг товаров, они отправятся на другой̆ склад и начнут разгружать товары.

Напишите программу на Java, используя многопоточность, которая реализует данную ситуацию.

Выполнение:

```
package lab7;


public class Task1 {

    private static int[] array;
```

```
    private static long sum1 = 0;
    private static long sum2 = 0;

    public static void main(String[] args) {
        System.out.println("=== Задание 1: Сумма элементов массива с двумя потоками ===\n");

        // Создаем массив
        array = new int[1000];
        for (int i = 0; i < array.length; i++) {
            array[i] = i + 1;
        }

        System.out.println("Массив из " + array.length + " элементов создан");
        System.out.println("Элементы: 1, 2, 3, ..., " + array.length);

        // Создаем два потока для вычисления суммы по половинкам
        Thread thread1 = new Thread(new SumCalculator(0, array.length / 2, 1));
        Thread thread2 = new Thread(new SumCalculator(array.length / 2, array.length, 2));
```

```
// Запускаем потоки
long startTime = System.currentTimeMillis();
thread1.start();
thread2.start();

// Ждем завершения обоих потоков
try {
    thread1.join();
    thread2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

long endTime = System.currentTimeMillis();

// Складываем результаты в главном потоке
long totalSum = sum1 + sum2;

System.out.println("\n--- Результаты ---");
System.out.println("Сумма первой половины (поток 1): " + sum1);
```

```
        System.out.println("Сумма второй половины (поток 2): " + sum2);
        System.out.println("Общая сумма: " + totalSum);
        System.out.println("Время выполнения: " + (endTime - startTime) + " мс");

        // Проверка правильности (формула суммы арифметической прогрессии)
        long expectedSum = (long) array.length * (array.length + 1) / 2;
        System.out.println("Ожидаемая сумма: " + expectedSum);
        System.out.println("Результат " + (totalSum == expectedSum ? "ВЕРЕН" : "НЕВЕРЕН"));
    }

    static class SumCalculator implements Runnable {
        private int start;
        private int end;
        private int threadNumber;

        public SumCalculator(int start, int end, int threadNumber) {
            this.start = start;
            this.end = end;
```

```
        this.threadNumber = threadNumber;
    }

    @Override
    public void run() {
        long partialSum = 0;
        System.out.println("Поток " + threadNumber + " начал работу (индексы " + start + "-" + (end - 1) + ")");

        for (int i = start; i < end; i++) {
            partialSum += array[i];
        }

        if (threadNumber == 1) {
            sum1 = partialSum;
        } else {
            sum2 = partialSum;
        }

        System.out.println("Поток " + threadNumber + " завершил работу. Сумма = " + partialSum);
```

```
        }
    }
}

package lab7;

import java.util.Random;

public class Task2 {
    private static int[][] matrix;
    private static int[] rowMaxValues;
    private static final int ROWS = 5;
    private static final int COLS = 10;

    public static void main(String[] args) {
        System.out.println("=== Задание 2: Поиск наибольшего элемента в матрице ===\n");

        // Создаем матрицу и заполняем случайными числами
        matrix = new int[ROWS][COLS];
        rowMaxValues = new int[ROWS];
```

```
Random random = new Random();


System.out.println("Матрица " + ROWS + "x" + COLS + ":");
for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
        matrix[i][j] = random.nextInt(100);
        System.out.printf("%4d", matrix[i][j]);
    }
    System.out.println();
}


// Создаем потоки для каждой строки
Thread[] threads = new Thread[ROWS];
for (int i = 0; i < ROWS; i++) {
    threads[i] = new Thread(new MaxFinder(i));
}


// Запускаем все потоки
long startTime = System.currentTimeMillis();
for (Thread thread : threads) {
```

```
    thread.start();
}

// Ждем завершения всех потоков
try {
    for (Thread thread : threads) {
        thread.join();
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}

long endTime = System.currentTimeMillis();

// Сравниваем результаты в главном потоке
System.out.println("\n--- Результаты поиска в каждой строке ---");
int globalMax = Integer.MIN_VALUE;
int maxRow = -1;

for (int i = 0; i < ROWS; i++) {
```

```
            System.out.println("Строка " + i + ": максимальный элемент = " +
rowMaxValues[i]);
            if (rowMaxValues[i] > globalMax) {
                globalMax = rowMaxValues[i];
                maxRow = i;
            }
        }

        System.out.println("\n--- Итоговый результат ---");
        System.out.println("Наибольший элемент в матрице: " + globalMax);
        System.out.println("Найден в строке: " + maxRow);
        System.out.println("Время выполнения: " + (endTime - startTime) + "
мс");
    }

    static class MaxFinder implements Runnable {
        private int rowIndex;

        public MaxFinder(int rowIndex) {
            this.rowIndex = rowIndex;
        }
```

```
        @Override
        public void run() {
            System.out.println("Поток для строки " + rowIndex + " начал работу");

            int max = Integer.MIN_VALUE;
            for (int j = 0; j < COLS; j++) {
                if (matrix[rowIndex][j] > max) {
                    max = matrix[rowIndex][j];
                }
            }

            rowMaxValues[rowIndex] = max;
            System.out.println("Поток для строки " + rowIndex + " завершил работу. Максимум = " + max);
        }
    }
}
```

```
package lab7;
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.*;

public class Task3 {
    private static final int MAX_WEIGHT_PER_TRIP = 150;
    private static final int NUM_WORKERS = 3;
    private static List<Integer> warehouse = new ArrayList<>();

    public static void main(String[] args) {
        System.out.println("=== Задание 3: Склад с грузчиками ===\n");

        // Создаем товары на складе с случайным весом
        Random random = new Random();
        int totalGoods = 20;
        System.out.println("Товары на складе:");
        for (int i = 0; i < totalGoods; i++) {
            int weight = random.nextInt(50) + 10; // вес от 10 до 59 кг
```

```
            warehouse.add(weight);
            System.out.println("Товар " + (i + 1) + ": " + weight + " кг");
        }

        int totalWeight = warehouse.stream().mapToInt(Integer::intValue).sum();
        System.out.println("\nОбщий вес товаров: " + totalWeight + " кг");
        System.out.println("Максимальный вес за одну поездку: " +
MAX_WEIGHT_PER_TRIP + " кг");
        System.out.println("Количество грузчиков: " + NUM_WORKERS +
"\n");

        // Создаем ExecutorService с пулом из 3 потоков (грузчиков)
        ExecutorService executor =
Executors.newFixedThreadPool(NUM_WORKERS);
        CompletionService<WorkerResult> completionService = new
ExecutorCompletionService<>(executor);

        int tripNumber = 1;

        // Пока есть товары на складе
        while (!warehouse.isEmpty()) {
            System.out.println("--- Поездка #" + tripNumber + " ---");
```

```
        List<Integer> currentLoad = new ArrayList<>();
        int currentWeight = 0;

        // Собираем товары до достижения лимита 150 кг
        synchronized (warehouse) {
            while (!warehouse.isEmpty() && currentWeight <
MAX_WEIGHT_PER_TRIP) {
                int nextWeight = warehouse.get(0);
                if (currentWeight + nextWeight <= MAX_WEIGHT_PER_TRIP) {
                    warehouse.remove(0);
                    currentLoad.add(nextWeight);
                    currentWeight += nextWeight;
                } else {
                    break;
                }
            }
        }

        if (currentLoad.isEmpty()) {
            // Если товар слишком тяжелый, берем его отдельно
```

```
            synchronized (warehouse) {
                if (!warehouse.isEmpty()) {
                    int heavyItem = warehouse.remove(0);
                    currentLoad.add(heavyItem);
                    currentWeight = heavyItem;
                }
            }
        }

        System.out.println("Собрано товаров: " + currentLoad.size() + " шт., общий вес: " + currentWeight + " кг");

        // Создаем задачи для грузчиков
        int itemsPerWorker = currentLoad.size() / NUM_WORKERS;
        int remainingItems = currentLoad.size() % NUM_WORKERS;

        int startIndex = 0;
        for (int i = 0; i < NUM_WORKERS; i++) {
            int itemsForThisWorker = itemsPerWorker + (i < remainingItems ? 1 : 0);
```

```
            if (itemsForThisWorker > 0) {
                List<Integer> workerLoad = currentLoad.subList(startIndex,
startIndex + itemsForThisWorker);
                completionService.submit(new Worker(i + 1, new
ArrayList<>(workerLoad), tripNumber));
                startIndex += itemsForThisWorker;
            }
        }

        // Получаем результаты выполнения задач
        int activeWorkers = Math.min(NUM_WORKERS, currentLoad.size());
        for (int i = 0; i < activeWorkers; i++) {
            try {
                Future<WorkerResult> future = completionService.take();
                WorkerResult result = future.get();
                System.out.println("  -> " + result.getMessage());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }
```

```
            System.out.println("Поездка #" + tripNumber + " завершена. Товары доставлены.\n");
            tripNumber++;
        }

        // Завершаем работу ExecutorService
        executor.shutdown();
        try {
            if (!executor.awaitTermination(5, TimeUnit.SECONDS)) {
                executor.shutdownNow();
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
        }

        System.out.println("=== Все товары перенесены! ===");
        System.out.println("Всего поездок: " + (tripNumber - 1));
    }

    static class Worker implements Callable<WorkerResult> {
        private int workerId;
```

```
    private List<Integer> items;

    public Worker(int workerId, List<Integer> items, int tripNumber) {
        this.workerId = workerId;
        this.items = items;
    }

    @Override
    public WorkerResult call() throws Exception {
        int totalWeight = items.stream().mapToInt(Integer::intValue).sum();

        // Имитация переноски товаров (время зависит от веса)
        Thread.sleep(totalWeight * 10);

        String message = "Грузчик " + workerId + " перенес " + items.size() +
                " товар(ов) весом " + totalWeight + " кг";

        return new WorkerResult(workerId, totalWeight, message);
    }
}
```

```
static class WorkerResult {
    private int workerId;
    private int weight;
    private String message;

    public WorkerResult(int workerId, int weight, String message) {
        this.workerId = workerId;
        this.weight = weight;
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public int getWorkerId() {
        return workerId;
    }
```

```
        public int getWeight() {
            return weight;
        }
    }
}
```

```
❯ java -cp bin lab7.Task1
=== Задание 1: Сумма элементов массива с двумя потоками ===

Массив из 1000 элементов создан
Элементы: 1, 2, 3, ..., 1000
Поток 1 начал работу (индексы 0-499)
Поток 2 начал работу (индексы 500-999)
Поток 2 завершил работу. Сумма = 375250
Поток 1 завершил работу. Сумма = 125250

--- Результаты ---
Сумма первой половины (поток 1): 125250
Сумма второй половины (поток 2): 375250
Общая сумма: 500500
Время выполнения: 6 мс
Ожидаемая сумма: 500500
Результат ВЕРЕН
```

```
❯ java -cp bin lab7.Task2
=== Задание 2: Поиск наибольшего элемента в матрице ===

Матрица 5x10:
  98  23  73  51  85  94  17  16   9  53
  54  32  27  34  95  31  32  85   7  64
  20  43  93  13  57  21  62  26   8  23
  38  42  61  38  62  35  20  69  93  15
  83  55  24  79  68  66  36  44  21  59
Поток для строки 1 начал работу
Поток для строки 3 начал работу
Поток для строки 0 начал работу
Поток для строки 4 начал работу
Поток для строки 2 начал работу
Поток для строки 3 завершил работу. Максимум = 93
Поток для строки 2 завершил работу. Максимум = 93
Поток для строки 0 завершил работу. Максимум = 98
Поток для строки 1 завершил работу. Максимум = 95
Поток для строки 4 завершил работу. Максимум = 83

--- Результаты поиска в каждой строке ---
Строка 0: максимальный элемент = 98
Строка 1: максимальный элемент = 95
Строка 2: максимальный элемент = 93
Строка 3: максимальный элемент = 93
Строка 4: максимальный элемент = 83

--- Итоговый результат ---
Наибольший элемент в матрице: 98
Найден в строке: 0
Время выполнения: 5 мс
```

```
❯ java -cp bin lab7.Task3
=== Задание 3: Склад с грузчиками ===

Товары на складе:
Товар 1: 37 кг
Товар 2: 14 кг
Товар 3: 40 кг
Товар 4: 56 кг
Товар 5: 48 кг
Товар 6: 12 кг
Товар 7: 15 кг
Товар 8: 10 кг
Товар 9: 39 кг
Товар 10: 33 кг
Товар 11: 46 кг
Товар 12: 24 кг
Товар 13: 29 кг
Товар 14: 43 кг
Товар 15: 24 кг
Товар 16: 16 кг
Товар 17: 50 кг
Товар 18: 25 кг
Товар 19: 38 кг
Товар 20: 25 кг

Общий вес товаров: 624 кг
Максимальный вес за одну поездку: 150 кг
Количество грузчиков: 3

--- Поездка #1 ---
Собрано товаров: 4 шт., общий вес: 147 кг
  -> Грузчик 2 перенес 1 товар(ов) весом 40 кг
  -> Грузчик 1 перенес 2 товар(ов) весом 51 кг
  -> Грузчик 3 перенес 1 товар(ов) весом 56 кг
Поездка #1 завершена. Товары доставлены.

--- Поездка #2 ---
Собрано товаров: 5 шт., общий вес: 124 кг
  -> Грузчик 2 перенес 2 товар(ов) весом 25 кг
  -> Грузчик 3 перенес 1 товар(ов) весом 39 кг
  -> Грузчик 1 перенес 2 товар(ов) весом 60 кг
Поездка #2 завершена. Товары доставлены.

--- Поездка #3 ---
Собрано товаров: 4 шт., общий вес: 132 кг
  -> Грузчик 2 перенес 1 товар(ов) весом 24 кг
  -> Грузчик 3 перенес 1 товар(ов) весом 29 кг
  -> Грузчик 1 перенес 2 товар(ов) весом 79 кг
Поездка #3 завершена. Товары доставлены.

--- Поездка #4 ---
Собрано товаров: 4 шт., общий вес: 133 кг
  -> Грузчик 2 перенес 1 товар(ов) весом 16 кг
  -> Грузчик 3 перенес 1 товар(ов) весом 50 кг
  -> Грузчик 1 перенес 2 товар(ов) весом 67 кг
Поездка #4 завершена. Товары доставлены.

--- Поездка #5 ---
Собрано товаров: 3 шт., общий вес: 88 кг
  -> Грузчик 1 перенес 1 товар(ов) весом 25 кг
  -> Грузчик 3 перенес 1 товар(ов) весом 25 кг
  -> Грузчик 2 перенес 1 товар(ов) весом 38 кг
Поездка #5 завершена. Товары доставлены.

=== Все товары перенесены! ===
Всего поездок: 5
```

Контрольные вопросы:

1. Как реализуется многопоточность в Java?

Многопоточность в Java реализуется двумя основными способами: наследованием класса Thread или реализацией интерфейса Runnable. В первом случае создается класс-наследник Thread и переопределяется метод run(). Во втором случае создается класс, реализующий интерфейс Runnable, после чего объект этого класса передается в конструктор Thread. Для более гибкого управления потоками используется фреймворк java.util.concurrent с классами ExecutorService, ThreadPoolExecutor и другими.

2. Что такое поток?

Поток (thread) — это независимая последовательность выполнения инструкций внутри программы. Поток представляет собой легковесный процесс, который разделяет общее адресное пространство с другими потоками того же приложения. Каждый поток имеет собственный стек вызовов, счетчик команд и локальные переменные, но разделяет с другими потоками кучу (heap) и область методов.

3. Для чего нужно ключевое слово synchronized?

Ключевое слово synchronized используется для синхронизации доступа к разделяемым ресурсам и предотвращения состояния гонки (race condition). Оно гарантирует, что только один поток может выполнять синхронизированный блок кода или метод в определенный момент времени. Synchronized может применяться к методам или блокам кода, используя монитор объекта для обеспечения взаимного исключения.

4. Для чего нужно ключевое слово volatile?

Ключевое слово volatile обеспечивает видимость изменений переменной между потоками. Оно гарантирует, что значение переменной всегда читается из основной памяти, а не из кэша процессора, и все изменения немедленно записываются в основную память. Volatile предотвращает оптимизации компилятора, которые могли бы привести к устаревшим значениям переменной в разных потоках.

5. Зачем нужно синхронизировать потоки?

Синхронизация потоков необходима для предотвращения проблем, возникающих при конкурентном доступе к разделяемым ресурсам: состояния гонки (race condition), когда результат зависит от порядка выполнения потоков; потери обновлений данных; чтения несогласованных данных; взаимной блокировки (deadlock). Синхронизация обеспечивает корректность работы программы при параллельном выполнении.

6. Какие есть способы синхронизации потоков?

Основные способы синхронизации в Java: synchronized методы и блоки; explicit locks (ReentrantLock, ReadWriteLock); атомарные классы (AtomicInteger, AtomicReference); volatile переменные; семафоры (Semaphore); барьеры (CyclicBarrier, CountDownLatch); блокирующие очереди (BlockingQueue); методы wait(), notify(), notifyAll(); класс Condition для более гибкой синхронизации.

7. В чем разница между Thread и Runnable?

Thread — это класс, представляющий поток выполнения, наследование которого создает новый тип потока. Runnable — это функциональный интерфейс, определяющий задачу для выполнения в потоке. Основные различия: класс может реализовать Runnable и наследовать другой класс (множественное наследование невозможно для Thread); Runnable обеспечивает лучшее разделение задачи и механизма выполнения; Runnable можно использовать с ExecutorService и другими средствами управления потоками; подход с Runnable более гибкий и соответствует современным практикам.

8. Какие состояния может иметь поток? Опишите жизненный цикл потока.

Поток в Java может находиться в следующих состояниях: NEW (создан, но не запущен), RUNNABLE (выполняется или готов к выполнению), BLOCKED (заблокирован в ожидании монитора), WAITING (ожидает неопределенное время действия другого потока), TIMED_WAITING (ожидает определенное время), TERMINATED (завершен). Жизненный цикл: создание (NEW) → start() → RUNNABLE → выполнение → возможные переходы в BLOCKED/WAITING/TIMED_WAITING → возврат в RUNNABLE → завершение метода run() → TERMINATED.

9. Что такое daemon-поток? Как его создать?

Daemon-поток — это фоновый поток, который не препятствует завершению программы. JVM завершается, когда все пользовательские потоки завершены, не дожидаясь завершения daemon-потоков. Создается вызовом метода setDaemon(true) до запуска потока: thread.setDaemon(true); thread.start();. Daemon-потоки используются для вспомогательных задач (сборка мусора, финализация объектов).

10. Как принудительно остановить поток?

Метод stop() устарел и опасен, так как может оставить объекты в несогласованном состоянии. Правильный способ остановки потока: использование флага прерывания через метод interrupt() и проверка Thread.interrupted() или Thread.currentThread().isInterrupted() внутри потока; использование volatile boolean флага для управления циклом выполнения. Поток должен периодически проверять условие остановки и корректно завершать работу.

11. Как работает метод join()? Для чего он используется?

Метод join() заставляет вызывающий поток ждать завершения потока, на котором он вызван. Это блокирующая операция, которая приостанавливает выполнение текущего потока до тех пор, пока целевой поток не завершится. Используется для синхронизации потоков, когда необходимо дождаться результата работы другого потока. Существуют перегруженные версии с таймаутом: join(long millis).

12. Что такое «гонка данных» (race condition)?

Race condition — это ситуация, когда результат выполнения программы зависит от порядка или времени выполнения неконтролируемых событий в разных потоках. Возникает при конкурентном доступе к разделяемым данным, когда хотя бы одна операция изменяет данные. Приводит к непредсказуемому поведению программы и трудноуловимым ошибкам. Предотвращается синхронизацией доступа к разделяемым ресурсам.

13. Что такое deadlock? Как его избежать?

Deadlock (взаимная блокировка) — это ситуация, когда два или более потока бесконечно ждут друг друга, каждый удерживая ресурс, необходимый другому. Способы избежать: всегда захватывать блокировки в одном и том же порядке; использовать таймауты при попытке захвата

блокировки (tryLock с таймаутом); минимизировать время удержания блокировки; использовать высокоуровневые утилиты java.util.concurrent; применять алгоритмы обнаружения и разрешения deadlock.

14. Что такое wait(), notify() и notifyAll()? В каком классе они объявлены?

Это методы класса Object, используемые для межпоточного взаимодействия. wait() переводит поток в состояние ожидания и освобождает монитор объекта; notify() пробуждает один произвольный ожидающий поток; notifyAll() пробуждает все ожидающие потоки. Вызываются только внутри синхронизированного блока или метода на объекте-мониторе. После пробуждения поток должен повторно захватить монитор перед продолжением выполнения.

15. Что такое ThreadPool? Какие реализации ExecutorService есть в Java?

ThreadPool — это набор предварительно созданных потоков, готовых к выполнению задач, что позволяет избежать накладных расходов на создание и уничтожение потоков. Основные реализации ExecutorService: ThreadPoolExecutor (базовая реализация с настраиваемыми параметрами), ScheduledThreadPoolExecutor (для отложенного и периодического выполнения), ForkJoinPool (для рекурсивных задач). Фабричные методы Executors создают типовые конфигурации: newFixedThreadPool, newCachedThreadPool, newSingleThreadExecutor, newScheduledThreadPool, newWorkStealingPool.

Заключение:

В ходе выполнения лабораторной работы были изучены фундаментальные концепции многопоточного программирования в Java и получены практические навыки разработки параллельных приложений. Реализованы классические задачи параллельного программирования: вычисление суммы элементов массива с разделением на несколько потоков, поиск максимального элемента в матрице с использованием параллельной обработки строк, а также моделирование системы управления ресурсами на примере задачи о складе с грузчиками.

Освоены различные подходы к созданию потоков (Thread, Runnable), механизмы синхронизации (synchronized, Lock, Semaphore,

CountDownLatch, CyclicBarrier) и средства управления пулами потоков (ExecutorService). Получен опыт работы с современными инструментами параллельного программирования из пакета java.util.concurrent, включая CompletableFuture для асинхронного выполнения и ForkJoinPool для рекурсивных задач.

Практическая работа позволила глубже понять проблемы, возникающие при конкурентном выполнении (race condition, deadlock), и способы их решения. Приобретенные знания и навыки являются необходимой основой для разработки высокопроизводительных многопоточных приложений и эффективного использования современных многоядерных процессоров.

Ссылка на GitHub репозиторий: https://github.com/M1ke0-0/ITiP