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 БВТ2402

Косякова Олеся Дмитриевна

Москва, 2025

Цель работы: Освоение практического применения многопоточности Java для решения типовых задач обработки информации.

Задание:

1) Реализация многопоточной программы для вычисления сум мы элементов массива.

2) Реализация многопоточной программы для поиска наибольшего элемента в матрице.

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

Ход работы:

Задание 1.

Вариант 1.

Создаем класс описывающий наш поток, в параметры передаем массив, индекс начала и индекс конца. Далее переопределяем метод run, в нем будем считать сумму массива, и в геттере прописываем получившуюся сумму.

```java
class SumThread extends Thread {
    private int[] array;
    private int start;
    private int end;
    private int sum = 0;

    public SumThread(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    public void run() {
        for (int i = start; i < end; i++) {
            sum += array[i];
        }
    }

    public int getSum() {
        return sum;
    }
}
```

В основном классе создаем два потока и передаем им начальные параметры, далее запускаем их. Вызываем метод join, который помогает не продолжать выполнение главного потока пока все потоки не завершат работу, считаем сумму и выводим.

```
public class SumArrayThreads {
    Run | Debug
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5, 6};

        int mid = array.length / 2;

        SumThread thread1 = new SumThread(array, start: 0, mid);
        SumThread thread2 = new SumThread(array, mid, array.length);

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        int totalSum = thread1.getSum() + thread2.getSum();

        System.out.println("Общая сумма элементов массива: " + totalSum);
    }
}
```

Вариант 2.

Создаем класс описывающий поток и имплементируем интерфейс Callable с его помощью мы можем переопределить метод call, который будет нам возвращать значение в отличие от run.

```
public class SumTask implements Callable<Long> {
    private final int[] arr;
    private final int from;
    private final int to;

    public SumTask(int[] arr, int from, int to) {
        this.arr = arr;
        this.from = from;
        this.to = to;
    }

    @Override
    public Long call() {
        long sum = 0;
        for (int i = from; i < to; i++) {
            sum += arr[i];
        }
        return sum;
    }
}
```

Создаем основной класс, прописываем метод заполнения массива, далее создаем пул потоков фиксированного размера. Далее создаем коллекцию из объектов Future, которые представляют асинхронные задачи, массив из обещаний результатов. Далее в цикле прописываем формулы для вычисления индекса начала и конца, далее результат выполнения потока мы передаем в пул потоков и сохраняем обещание результата в коллекцию. Далее прописываем остановление работы пула потоков, при котором он не будет принимать новые задачи, а старые продолжит выполнять. Далее последовательно получаем результаты всех future и суммируем их.

```java
public class SumArrayExecutor {
    Run | Debug
    public static void main(String[] args) {

        int[] array = new int[2_000_000];
        for (int i = 0; i < array.length; i++) {
            array[i] = 1;
        }

        final int th_count = 4;

        ExecutorService executor = Executors.newFixedThreadPool(th_count);

        List<Future<Long>> results = new ArrayList<>();

        int partSize = array.length / th_count;

        for (int i = 0; i < th_count; i++) {
            int start = i * partSize;
            int end;
            if (i == th_count - 1) {
                end = array.length;
            } else {
                end = start + partSize;
            }

            results.add(executor.submit(new SumTask(array, start, end)));
        }

        executor.shutdown();

        long totalSum = 0;
        try {
            for (Future<Long> future : results) {
                totalSum += future.get();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("Общая сумма элементов массива = " + totalSum);
    }
}
```

Задание 2.

Вариант 1.

Переопределяем метод run, проходимся по значениям строки и сравниваем с максимальным на данный момент значением, прописываем геттер.

```java
public class MaxRowThread extends Thread {
    private final int[][] matrix;
    private final int rowIndex;
    private int maxInRow = 0;

    public MaxRowThread(int[][] matrix, int rowIndex) {
        this.matrix = matrix;
        this.rowIndex = rowIndex;
    }

    @Override
    public void run() {
        int[] row = matrix[rowIndex];
        for (int value : row) {
            if (value > maxInRow) {
                maxInRow = value;
            }
        }
    }

    public int getMaxInRow() {
        return maxInRow;
    }
}
```

Создаем матрицу, наполняем значениями, далее создаем потоки для каждой строки, запускаем каждый на поток, и для каждого прописываем join. Далее прописываем цикл, в котором получаем максимальное значение каждой строки и сравниваем его с максимальным значением на данный момент.

```java
public class MaxInMatrixThreads {
    Run | Debug
    public static void main(String[] args) {

        int rows = 10;
        int cols = 8;
        int[][] matrix = new int[rows][cols];

        Random rand = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = rand.nextInt(bound: 1000);
            }
        }

        System.out.println(x: "Матрица:");
        printMatrix(matrix);

        MaxRowThread[] threads = new MaxRowThread[rows];

        for (int i = 0; i < rows; i++) {
            threads[i] = new MaxRowThread(matrix, i);
            threads[i].start();
        }

        for (int i = 0; i < rows; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        int globalMax = 0;
        for (int i = 0; i < rows; i++) {
            int rowMax = threads[i].getMaxInRow();
            if (rowMax > globalMax) {
                globalMax = rowMax;
            }
        }

        System.out.println("Наибольший элемент во всей матрице = " + globalMax);
    }
```

Вариант 2.

Создаем пул потоков фиксированного размера, далее создаем список из объектов, представляющие асинхронные результаты. Далее результат работы потока добавляем в пул и сохраняем обещание результата в наш список.

```java
public class MaxInMatrixExecutor {
    public static void main(String[] args) {
        int rows = 10_000;
        int cols = 5_000;
        int[][] matrix = new int[rows][cols];

        Random rand = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = rand.nextInt(bound: 1_000_000);
            }
        }

        // Вставим максимум, чтобы точно знать ответ (для проверки)
        matrix[7532][1247] = 999_999;


        final int th_count = 8;
        try (ExecutorService executor = Executors.newFixedThreadPool(th_count)) {

            List<Future<Integer>> results = new ArrayList<>();

            int rowsPerThread = rows / th_count;

            for (int i = 0; i < th_count; i++) {
                int startRow = i * rowsPerThread;
                int endRow;

                if (i == th_count - 1) {
                    endRow = rows;
                } else {
                    endRow = startRow + rowsPerThread;
                }

                results.add(executor.submit(new MaxInPartTask(matrix, startRow, endRow)));
            }

            int globalMax = 0;

            for (Future<Integer> future : results) {
                int partMax = future.get();
                if (partMax > globalMax) {
                    globalMax = partMax;
                }
            }
```

Задание 3.

Создаем класс описывающий товар, в нашем случае только его вес.

```java
class Product {
    private int weight;

    public Product(int weight) {
        this.weight = weight;
    }

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

Далее класс описывающий наш склад. Создаем коллекцию, в которой будут храниться товары для разгрузки, переменная отвечающая за вес разгруженных товаров и максимальная вместительность. Далее прописываем синхронизированные методы, чтобы во время работы потока доступ к данным был только у одного. Метод взятия продукта если он не нул удаляем его из массива, далее добавляем массу этого продукта в переменную если она не больше 150 кг, иначе мы обнуляем переменную.

```
class Storage {
    private List<Product> products = new ArrayList<>();
    private int currentLoad = 0; // сколько сейчас грузчики собрали
    private final int MAX_CAPACITY = 150;

    public Storage(List<Product> products) {
        this.products = products;
    }

    public synchronized Product takeProduct() {
        if (products.isEmpty()) return null;
        return products.remove(index: 0);
    }

    public synchronized void addToLoad(int w) {
        currentLoad += w;
        System.out.println(Thread.currentThread().getName() +
                " взял " + w + " кг, сейчас собрано: " + currentLoad);

        if (currentLoad >= MAX_CAPACITY) {
            System.out.println(x: "150 кг собрано! Грузчики пошли разгружать");
            currentLoad = 0;
            System.out.println(x: "Грузчики вернулись");
        }
    }
}
```

Создаем класс описывающий поток работников. Пока мы можем взять товар мы его берем со склада (вызываем специальный метод), если товаров не осталось останавливаем цикл и далее вес нашего товара добавляем в переменную.

```java
class Worker extends Thread {
    private Storage storage;

    public Worker(Storage storage) {
        this.storage = storage;
    }

    @Override
    public void run() {
        while (true) {
            Product p = storage.takeProduct();
            if (p == null) {
                System.out.println(Thread.currentThread().getName() +
                        " — товаров больше нет");
                break;
            }
            storage.addToLoad(p.getWeight());
        }
    }
}
```

В основном методе мы создаем список товаров и заполняем его, создаем склад с продуктами и потоки для трех работников, далее запускаем наши потоки.

```java
public class Transfer {
    Run | Debug
    public static void main(String[] args) {

        List<Product> products = new ArrayList<>();

        // создаём 20 товаров с разным весом
        for (int i = 0; i < 20; i++) {
            products.add(new Product(10 + (int)(Math.random()*20)));
        }

        Storage storage = new Storage(products);

        Worker w1 = new Worker(storage);
        Worker w2 = new Worker(storage);
        Worker w3 = new Worker(storage);

        w1.setName(name: "Грузчик 1");
        w2.setName(name: "Грузчик 2");
        w3.setName(name: "Грузчик 3");

        w1.start();
        w2.start();
        w3.start();
    }
}
```

Вывод: В ходе лабораторной работы мы освоили многопоточность Java для решения типовых задач обработки информации.

Ссылка на репозиторий: https://github.com/swydiz/information_technology