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

Лабораторная работа №7

Многопоточность

Выполнил: студент группы ___БВТ2402___
Голиков Михаил Вячеславович
Руководитель: _Мосева Марина Сергеевна_

Москва, 2077

Цель работы

Целью лабораторной работы является изучение принципов многопоточного программирования в языке Java, освоение механизмов создания и управления потоками выполнения, а также приобретение практических навыков работы с классами ExecutorService, Thread и ForkJoinPool для решения задач параллельной обработки данных.

Индивидуальное задание

В рамках лабораторной работы необходимо реализовать следующие задачи:

1. **Задача 1**

   Реализация многопоточной программы для вычисления суммы элементов массива. Создать пул потоков с помощью класса ExecutorService и разделить массив на равные части, каждую из которых будет обрабатывать отдельный поток. После завершения работы всех потоков результаты будут складываться в главном потоке.

2. **Задача 2**

   Реализация многопоточной программы для поиска наибольшего элемента в матрице. Создать несколько потоков, каждый из которых будет обрабатывать свою строку матрицы. После завершения работы всех потоков результаты будут сравниваться в главном потоке для нахождения наибольшего элемента.

3. **Задача 3**

   Реализация программы для моделирования работы грузчиков на складе. Три грузчика переносят товары одновременно, при этом суммарный вес товаров за одну итерацию не должен превышать 150 кг. Использовать ForkJoinPool для разделения склада на подзадачи и обработки этих подзадач.

(Вариант 6).

Основная часть

В процессе выполнения лабораторной работы были реализованы три программы, демонстрирующие различные подходы к многопоточному программированию:

1. **ArraySum** – программа для вычисления максимального элемента в большом массиве с использованием ExecutorService. Массив разделяется на 4 части, каждая из которых обрабатывается в отдельном потоке. Результаты объединяются в главном потоке.
2. **MatrixCalc** – программа для поиска максимального элемента в матрице с использованием наследования от класса Thread. Каждый поток обрабатывает отдельную строку матрицы, после чего результаты сравниваются в главном потоке.
3. **WarehouseForkJoin** – программа для моделирования работы грузчиков на складе с использованием ForkJoinPool. Задача по переносу товаров разделяется на подзадачи, которые выполняются параллельно с соблюдением ограничения по весу.

```
package Lab_7;
// Var 2

import java.util.Collection;
import java.util.List;
import java.util.concurrent.*;


public class ArraySum {
    public static void main(String[] args) throws Exception {
        final int ARRAY_SIZE = 100_000_000;

        long[] myarr = new long[ARRAY_SIZE];

        for (int i = 0; i < ARRAY_SIZE; i++) {
            myarr[i] = (long) i * i;
        }

        Callable<Long> task1 = createTask(myarr, 1, ARRAY_SIZE);
        Callable<Long> task2 = createTask(myarr, 2, ARRAY_SIZE);
        Callable<Long> task3 = createTask(myarr, 3, ARRAY_SIZE);
```

```
        Callable<Long> task4 = createTask(myarr, 4, ARRAY_SIZE);

        ExecutorService executor = Executors.newFixedThreadPool(4);

        Future<Long> maxFirst = executor.submit(task1);
        Future<Long> maxSecond = executor.submit(task2);
        Future<Long> maxThird = executor.submit(task3);
        Future<Long> maxFourth = executor.submit(task4);

        long max1 = maxFirst.get();
        long max2 = maxSecond.get();
        long max3 = maxThird.get();
        long max4 = maxFourth.get();

        long overallMax = Math.max(Math.max(max1, max2), Math.max(max3,
max4));

        System.out.println("Maximum threaded: " + overallMax);

        executor.shutdown();

    }

    private static Callable<Long> createTask(long[] array, int part, int
arraySize) {
        return () -> {
            int start, end;

            switch (part) {
                case 1:
                    start = 0;
                    end = arraySize / 4;
                    break;
                case 2:
                    start = arraySize / 4;
                    end = arraySize / 2;
                    break;
                case 3:
                    start = arraySize / 2;
                    end = arraySize * 3 / 4;
                    break;
                default:
                    start = arraySize * 3 / 4;
                    end = arraySize;
                    break;
            }

            long maxi = 0;
            for (int i = start; i < end; i++) {
                if (maxi < array[i]) maxi = array[i];
            }

            System.out.println("Part " + part + " max: " + maxi + " (range: "
+ start + "-" + (end-1) + ")");
            return maxi;
        };
    }
}
```

Элемент 1 — Код программы 1

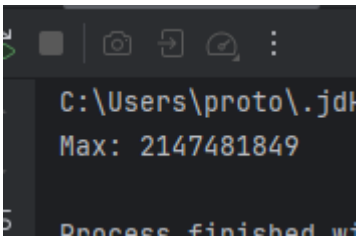
```
C:\Users\proto\.jdks\openjdk-25\bin\java.exe "-javaagent:
Part 3 max: 5624999850000001 (range: 50000000-74999999)
Part 4 max: 9999999800000001 (range: 75000000-99999999)
Part 2 max: 2499999900000001 (range: 25000000-49999999)
Part 1 max: 624999950000001 (range: 0-24999999)
Maximum threaded: 9999999800000001

Process finished with exit code 0
```

Элемент 2 — Результат программы 1

```java
package Lab_7;

public class MatrixCalc extends Thread{

    public int[][] matrixInside;

    public long maxi;
    public int row_num;

    public MatrixCalc(int row, int[][] matrix) {
        this.row_num = row;
        this.matrixInside = matrix;
    }

    @Override
    public void run(){
        for(int elem : matrixInside[row_num]){
            if(elem > maxi) maxi = elem;
        }
    }


    public static void main(String[] args) throws InterruptedException{
        final int ROW_SIZE = 1_000_000;

        int[][] matrix = new int[3][ROW_SIZE];
        for (int i = 0; i < ROW_SIZE; i++){
            matrix[0][i] = i;
            matrix[1][i] = i*i;
            matrix[2][i] = i*i*i;
        }


        MatrixCalc calc1 = new MatrixCalc(0, matrix);
        MatrixCalc calc2 = new MatrixCalc(1, matrix);
        MatrixCalc calc3 = new MatrixCalc(2, matrix);


        calc1.start();
        calc2.start();
        calc3.start();

        calc1.join();
        calc2.join();
        calc3.join();
```

```
        long overallMax = Math.max(Math.max(calc1.maxi, calc2.maxi),
calc3.maxi);
        System.out.println("Max: " + overallMax);
    }
}
```

Элемент 3 — Код программы 2



Элемент 4 — Результат программы 2

```
package Lab_7;

import java.util.*;
import java.util.concurrent.*;

public class WarehouseForkJoin {
    private static final int MAX_WEIGHT_PER_TRIP = 150;
    private static final int NUM_LOADERS = 3;


    static class Item {
        private final String name;
        private final int weight;

        public Item(String name, int weight) {
            this.name = name;
            this.weight = weight;
        }

        public int getWeight() { return weight; }
        public String getName() { return name; }

        @Override
        public String toString() {
            return name + "(" + weight + "кг)";
        }
    }


    static class LoadResult {
        private final List<Item> loadedItems = new ArrayList<>();
        private int totalWeight = 0;

        public boolean canAddItem(Item item) {
            return totalWeight + item.getWeight() <= MAX_WEIGHT_PER_TRIP;
        }

        public void addItem(Item item) {
            loadedItems.add(item);
            totalWeight += item.getWeight();
        }
```

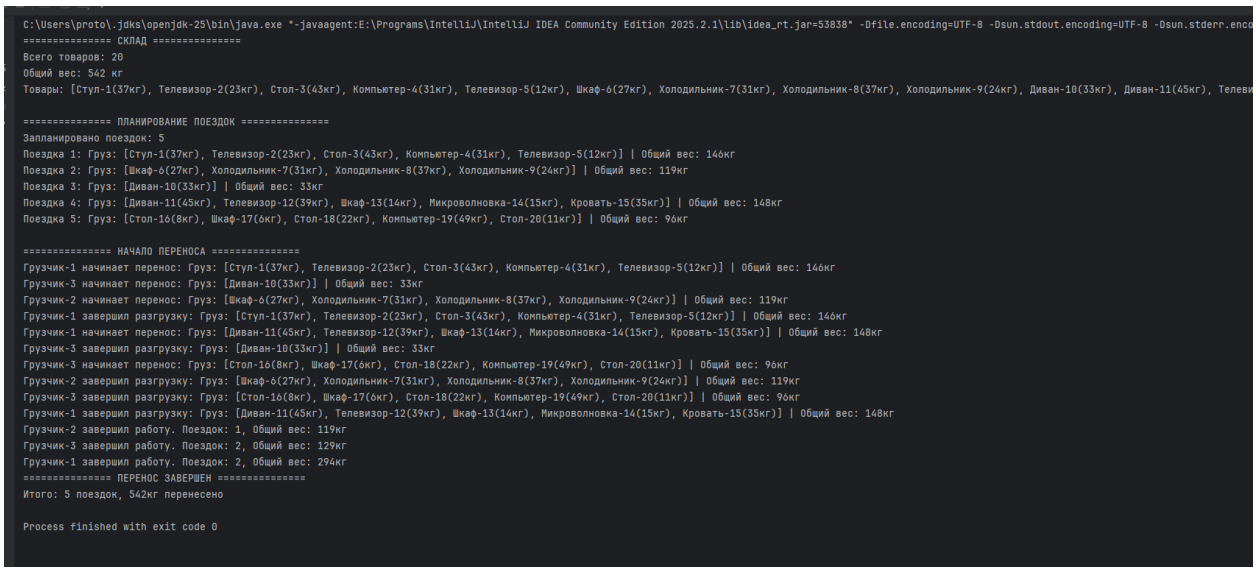
```
        public int getTotalWeight() { return totalWeight; }
        public List<Item> getLoadedItems() { return loadedItems; }

        @Override
        public String toString() {
            return "Груз: " + loadedItems + " | Общий вес: " + totalWeight +
"кг";
        }
    }


    static class WarehouseTask extends RecursiveTask<List<LoadResult>> {
        private final List<Item> items;
        private final int start;
        private final int end;
        private static final int THRESHOLD = 10; // Порог деления

        public WarehouseTask(List<Item> items, int start, int end) {
            this.items = items;
            this.start = start;
            this.end = end;
        }

        @Override
        protected List<LoadResult> compute() {
            if (end - start <= THRESHOLD) {
                return processSequentially();

            } else {

                int mid = start + (end - start) / 2;
                WarehouseTask leftTask = new WarehouseTask(items, start,
mid);
                WarehouseTask rightTask = new WarehouseTask(items, mid, end);

                leftTask.fork(); // Запускаем левую задачу асинхронно
                List<LoadResult> rightResult = rightTask.compute(); //
Выполняем правую задачу
                List<LoadResult> leftResult = leftTask.join(); // Ждем
результат левой задачи

                return mergeResults(leftResult, rightResult);
            }
        }

        private List<LoadResult> processSequentially() {
            List<LoadResult> allTrips = new ArrayList<>();
            LoadResult currentTrip = new LoadResult();

            for (int i = start; i < end; i++) {
                Item item = items.get(i);

                if (currentTrip.canAddItem(item)) {
                    currentTrip.addItem(item);
                } else {

                    if (currentTrip.getTotalWeight() > 0) {
                        allTrips.add(currentTrip);
                    }
                    currentTrip = new LoadResult();
                    currentTrip.addItem(item);
                }
```

```
            }

            // Последняя
            if (currentTrip.getTotalWeight() > 0) {
                allTrips.add(currentTrip);
            }

            return allTrips;
        }

        private List<LoadResult> mergeResults(List<LoadResult> left,
List<LoadResult> right) {
            if (left.isEmpty()) return right;
            if (right.isEmpty()) return left;

            List<LoadResult> merged = new ArrayList<>(left);

            // Последняя поездка left + первая поездка right
            LoadResult lastLeft = left.get(left.size() - 1);
            LoadResult firstRight = right.get(0);


            if (lastLeft.getTotalWeight() + firstRight.getTotalWeight() <=
MAX_WEIGHT_PER_TRIP) {

                LoadResult mergedTrip = new LoadResult();
                for (Item item : lastLeft.getLoadedItems()) {
                    mergedTrip.addItem(item);
                }
                for (Item item : firstRight.getLoadedItems()) {
                    mergedTrip.addItem(item);
                }


                merged.remove(merged.size() - 1);
                merged.add(mergedTrip);
                merged.addAll(right.subList(1, right.size()));
            } else {

                merged.addAll(right);
            }

            return merged;
        }
    }


    static class Loader extends Thread {
        private final int id;
        private final BlockingQueue<LoadResult> tripQueue;
        private int tripsCompleted = 0;
        private int totalWeightCarried = 0;

        public Loader(int id, BlockingQueue<LoadResult> tripQueue) {
            this.id = id;
            this.tripQueue = tripQueue;
            setName("Грузчик-" + id);
        }

        @Override
        public void run() {
            try {
                while (true) {
```

```
                    // Берем поездку синхронно
                    LoadResult trip = tripQueue.poll(1, TimeUnit.SECONDS);

                    if (trip == null) {
                        break;
                    }


                    System.out.println(getName() + " начинает перенос: " +
trip);
                    Thread.sleep(500 + new Random().nextInt(1000));

                    tripsCompleted++;
                    totalWeightCarried += trip.getTotalWeight();

                    System.out.println(getName() + " завершил разгрузку: " +
trip);
                }

                System.out.println(getName() + " завершил работу. " +
                        "Поездок: " + tripsCompleted +
                        ", Общий вес: " + totalWeightCarried + "кг");

            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    public static void main(String[] args) {

        List<Item> warehouse = generateWarehouse(50);

        System.out.println("=============== СКЛАД ===============");
        System.out.println("Всего товаров: " + warehouse.size());
        int totalWeightBefore = 0;
        for (Item item : warehouse) {
            totalWeightBefore += item.getWeight();
        }
        System.out.println("Общий вес: " + totalWeightBefore + " кг");

        System.out.println("Товары: " + warehouse);
        System.out.println();


        ForkJoinPool forkJoinPool = new ForkJoinPool();
        WarehouseTask mainTask = new WarehouseTask(warehouse, 0,
warehouse.size());

        System.out.println("=============== ПЛАНИРОВАНИЕ ПОЕЗДОК
===============");
        List<LoadResult> allTrips = forkJoinPool.invoke(mainTask);

        System.out.println("Запланировано поездок: " + allTrips.size());
        for (int i = 0; i < allTrips.size(); i++) {
            System.out.println("Поездка " + (i + 1) + ": " +
allTrips.get(i));
        }
        System.out.println();


        BlockingQueue<LoadResult> tripQueue = new
LinkedBlockingQueue<>(allTrips);
```

```
        System.out.println("=============== НАЧАЛО ПЕРЕНОСА
===============");


        List<Loader> loaders = new ArrayList<>();
        for (int i = 1; i <= NUM_LOADERS; i++) {
            Loader loader = new Loader(i, tripQueue);
            loaders.add(loader);
            loader.start();
        }


        for (Loader loader : loaders) {
            try {
                loader.join();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        System.out.println("=============== ПЕРЕНОС ЗАВЕРШЕН
===============");

        int totalTrips = loaders.stream().mapToInt(l ->
l.tripsCompleted).sum();
        int totalWeight = loaders.stream().mapToInt(l ->
l.totalWeightCarried).sum();

        System.out.println("Итого: " + totalTrips + " поездок, " +
totalWeight + "кг перенесено");

        forkJoinPool.shutdown();
    }


    private static List<Item> generateWarehouse(int itemCount) {
        List<Item> warehouse = new ArrayList<>();
        Random random = new Random();
        String[] itemNames = {"Холодильник", "Телевизор", "Стиральная
машина", "Диван",
                "Стол", "Стул", "Кровать", "Шкаф", "Микроволновка",
"Компьютер"};

        for (int i = 1; i <= itemCount; i++) {
            String name = itemNames[random.nextInt(itemNames.length)] + "-" +
i;
            int weight = 5 + random.nextInt(46); // Вес от 5 до 50 кг
            warehouse.add(new Item(name, weight));
        }

        return warehouse;
    }
}
```

Элемент 5 — Код программы 3



Элемент 6 — Результат программы 3

Заключение

В ходе выполнения лабораторной работы были успешно изучены и применены на практике основные механизмы многопоточного программирования в Java. Были реализованы программы, демонстрирующие:

- использование ExecutorService для создания пула потоков и параллельной обработки данных;
- создание и управление потоками через наследование от класса Thread;
- применение ForkJoinPool для рекурсивного разделения задач на подзадачи;
- синхронизацию работы потоков и объединение результатов.

Работа позволила закрепить навыки проектирования многопоточных приложений, понимание принципов параллельных вычислений и методов оптимизации производительности программ.

Github: https://github.com/Prototype721/Java