МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ

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

По дисциплине «Информационные технологии и программирование»

Выполнил: Студент группы

БПИ 2301

Антонова Ирина

Москва

2024

Цель:

Изучение и применение многопоточности в Java

Задание:

Задание 1. Реализация многопоточной программы для вычисления суммы элементов массива.

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

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

Вариант 1. 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.

Задание 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 одновременно, но суммарный вес товаров, которые они переносят, не может превышать 150 кг. Как только грузчики соберут 150 кг товаров, они отправятся на другой склад и начнут разгружать товары. Напишите программу на Java, используя многопоточность, которая реализует данную ситуацию.

Вариант 4: Использование Semaphore: Используйте семафоры для ограничения доступа к складу и контроля над весом товаров.

Ход работы:

Откроем папку LR7 в программе VSCode и начнем создавать файлы, необходимые для выполнения заданий.

Задание 1:

Создадим в папке файл ArraySum.java для реализации методов.

Объявление класса:

```
static class SumArray implements Runnable {
```

Класс SumArray реализует интерфейс Runnable, что позволяет его экземплярам использоваться в качестве задач для потоков.

Поля класса:

```
        private final int[] array;
        private final int start;
        private final int end;
        private int partialSum;
```

array: Массив, элементы которого будут суммироваться.

start: Начальный индекс для суммирования.

end: Конечный индекс для суммирования.

partialSum: Частичная сумма элементов массива в заданном диапазоне.

Конструктор:

```
        public SumArray(int[] array, int start, int end) {
            this.array = array;
            this.start = start;
            this.end = end;
            this.partialSum = 0;
        }
```

Конструктор инициализирует поля класса.

Метод run:

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

Метод run выполняет суммирование элементов массива в заданном диапазоне и сохраняет результат в partialSum.

Метод getPartialSum:

```
        public int getPartialSum() {
            return partialSum;
        }
```

Метод возвращает частичную сумму.

Метод main

Инициализация массива:

```
    public static void main(String[] args) throws InterruptedException {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Создается массив целых чисел.

Разделение массива на две части:

```
int mid = array.length / 2;
```

Определяется середина массива.

Создание объектов SumArray:

```
        SumArray firstHalf = new SumArray(array, 0, mid);
        SumArray secondHalf = new SumArray(array, mid, array.length);
```

Создаются два объекта SumArray, каждый из которых отвечает за суммирование половины массива.

Создание и запуск потоков:

```
        Thread thread1 = new Thread(firstHalf);
        Thread thread2 = new Thread(secondHalf);

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

Создаются два потока, каждый из которых выполняет задачу суммирования своей части массива.

Ожидание завершения потоков:

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

Основной поток ожидает завершения выполнения обоих потоков.

Вычисление общей суммы:

```
        int totalSum = firstHalf.getPartialSum() + secondHalf.getPartialSum();
```

Общая сумма вычисляется как сумма частичных сумм.

Вывод результата:

```
System.out.println("Сумма: " + totalSum);
```

Результат выводится на экран.

```
PS C:\Users\ira\OneDrive\Desktop\work\ИТИП\LR7> java ArraySum.java
Сумма: 45
PS C:\Users\ira\OneDrive\Desktop\work\ИТИП\LR7>
```

Задание 2:

Класс BiggestNumber

```
public class BiggestNumber {
```

Вложенный класс FindMaxInRow

Объявление класса:

Класс FindMaxInRow реализует интерфейс Runnable, что позволяет его экземплярам использоваться в качестве задач для потоков.

Поля класса:

```
    static class FindMaxInRow implements Runnable {
        private final int[] row;
        private final int rowIndex;
        private int max;
```

row: Строка матрицы, в которой будет искаться максимальный элемент.

rowIndex: Индекс строки в матрице.

max: Максимальный элемент в строке.

Конструктор:

```
        public FindMaxInRow(int[] row, int rowIndex) {
            this.row = row;
            this.rowIndex = rowIndex;
            this.max = Integer.MIN_VALUE;
        }
```

Конструктор инициализирует поля класса.

Метод run:

```
        @Override
        public void run() {
            max = findMaxInRow(row);
            System.out.println("Максимум в строке " + rowIndex + ": " + max);
```

Метод run выполняет поиск максимального элемента в строке и сохраняет результат в max.

Метод getMax:

```
        public int getMax() {
            return max;
        }
```

Метод возвращает максимальный элемент в строке.

Метод main

Инициализация матрицы:

```
    public static void main(String[] args) {
        int[][] matrix = {
            {3, 8, 2, 10},
            {5, 1, 7, 6},
            {12, 14, 4, 9},
            {11, 15, 13, 0}
        };
```

Создается матрица целых чисел.

Определение количества строк:

```
 int rows = matrix.length;
```

Создание массивов для хранения результатов и потоков:

```
        int[] maxInRows = new int[rows];
        Thread[] threads = new Thread[rows];
        FindMaxInRow[] tasks = new FindMaxInRow[rows];
```

maxInRows: Массив для хранения максимальных элементов каждой строки.

threads: Массив для хранения потоков.

tasks: Массив для хранения экземпляров FindMaxInRow.

Создание и запуск потоков:

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

Для каждой строки матрицы создается экземпляр FindMaxInRow, который затем используется для создания и запуска потока.

Ожидание завершения потоков и получение результатов:

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

Основной поток ожидает завершения выполнения всех потоков и затем получает максимальные элементы из каждой строки.

Вычисление общего максимума:

```
int overallMax = findMaxInRow(maxInRows);
```

Общий максимум вычисляется как максимальный элемент среди максимальных элементов всех строк.

Вывод результата:

```
System.out.println("Наибольший элемент в матрице: " + overallMax + " \nДлина матрицы: " + rows);
```

Результат выводится на экран.

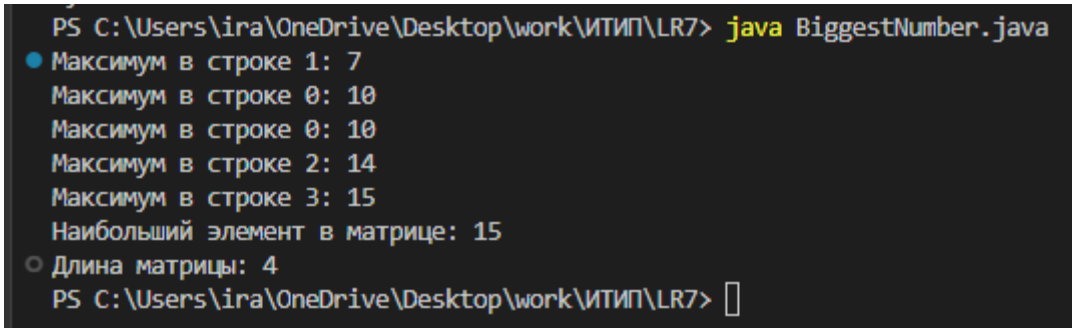
Метод findMaxInRow

Объявление метода:

Метод принимает строку матрицы и возвращает максимальный элемент в этой строке.

```
public static int findMaxInRow(int[] row) {
        int max = row[0];
        for (int num : row) {
            if (num > max) {
                max = num;
            }
        }
        return max;
    }
```

Метод проходит по всем элементам строки и находит максимальный элемент.



Задание 3:

Семафор — это механизм синхронизации, который используется для управления доступом к ресурсам в многопоточных приложениях. Он позволяет ограничить количество потоков, которые могут одновременно использовать определенный ресурс. Семафоры особенно полезны в ситуациях, когда необходимо ограничить количество одновременно выполняющихся операций или доступ к ограниченному ресурсу.

Основные концепции семафора

1. Счетчик (Counter): Семафор имеет внутренний счетчик, который указывает количество доступных разрешений (permits). Когда семафор создается, счетчик инициализируется определенным значением.
2. Операции:
    - Acquire (Захват): Поток пытается захватить разрешение. Если счетчик больше нуля, он уменьшается на единицу, и поток продолжает выполнение. Если счетчик равен нулю, поток блокируется до тех пор, пока другой поток не освободит разрешение.
    - Release (Освобождение): Поток освобождает разрешение, увеличивая счетчик на единицу. Если есть блокированные потоки, один из них разблокируется и получает разрешение.

Пошаговое объяснение работы семафора в коде

1. Инициализация семафора:

private static final Semaphore semaphore = new Semaphore(3);

Семафор инициализируется с тремя разрешениями, что означает, что одновременно могут работать три грузчика.

2. Захват семафора:

semaphore.acquire();

Грузчик пытается захватить разрешение. Если разрешения доступны (счетчик больше нуля), счетчик уменьшается на единицу, и грузчик продолжает выполнение. Если разрешения недоступны (счетчик равен нулю), грузчик блокируется до тех пор, пока другой грузчик не освободит разрешение.

3. Освобождение семафора:

semaphore.release();

Грузчик освобождает разрешение, увеличивая счетчик на единицу. Если есть блокированные грузчики, один из них разблокируется и получает разрешение.

Преимущества использования семафора

1. Ограничение доступа: Семафор позволяет ограничить количество потоков, которые могут одновременно использовать определенный ресурс, что помогает избежать перегрузки и конфликтов.
2. Синхронизация: Семафор обеспечивает синхронизацию доступа к ресурсам, предотвращая одновременное использование ресурса несколькими потоками.
3. Гибкость: Семафоры могут использоваться для управления доступом к различным типам ресурсов, таким как файлы, сетевые соединения, базы данных и т.д.

Примеры использования семафора

1. Ограничение количества одновременных запросов к серверу: Семафор может использоваться для ограничения количества одновременных запросов к серверу, чтобы избежать перегрузки.
2. Управление доступом к ограниченному ресурсу: Семафор может использоваться для управления доступом к ограниченному ресурсу, таким как принтер или файл.
3. Синхронизация потоков: Семафор может использоваться для синхронизации потоков, чтобы обеспечить корректное выполнение операций в многопоточных приложениях.

Класс Warehouse

Поля класса

Константы и переменные:

```
public class Warehouse {
    private static final int MAX_WEIGHT = 150;
    private static final Semaphore semaphore = new Semaphore(3);
    private static int[] items = {50, 20, 30, 40, 10, 60, 70, 80, 90, 30};
    private static int currentIndex = 0;
    private static final Object lock = new Object();
```

MAX_WEIGHT: Максимальный вес, который может быть загружен одним грузчиком.

semaphore: Семафор, который ограничивает количество одновременно работающих грузчиков до 3.

items: Массив весов предметов, которые нужно загрузить.

currentIndex: Текущий индекс в массиве items, который используется для отслеживания, какой предмет будет загружен следующим.

lock: Объект для синхронизации доступа к currentIndex.

Метод main

Создание и запуск потоков:

```
    public static void main(String[] args) {
        Thread loader1 = new Thread(new Loader("Грузчик 1"));
        Thread loader2 = new Thread(new Loader("Грузчик 2"));
        Thread loader3 = new Thread(new Loader("Грузчик 3"));

        loader1.start();
        loader2.start();
        loader3.start();
    }
```

Создаются три потока, каждый из которых представляет грузчика. Потоки запускаются.

Вложенный класс Loader

Поля класса

Имя грузчика:

```
    static class Loader implements Runnable {
```

```
        private String name;
```
Имя грузчика, которое будет использоваться для вывода сообщений.

Конструктор

Инициализация имени грузчика:

```
        public Loader(String name) {
            this.name = name;
        }
```
Конструктор инициализирует имя грузчика.

Метод run

Основной цикл:

```
        @Override
        public void run() {
            while (true) {
                try {
                    semaphore.acquire();

                    int totalWeight = 0;
                    StringBuilder itemsToLoad = new StringBuilder();

                    while (true) {
                        int itemWeight;

                        synchronized (lock) {
                            if (currentIndex >= items.length) {
                                break;
                            }
                            itemWeight = items[currentIndex];
                            currentIndex++;
                        }

                        if (totalWeight + itemWeight <= MAX_WEIGHT) {
                            totalWeight += itemWeight;
                            itemsToLoad.append(itemWeight).append(" кг, ");
                        } else {

                            synchronized (lock) {
                                currentIndex--;
                            }
                            break;
                        }
                    }
```

```
                    if (totalWeight > 0) {
                        System.out.println(name + " загрузил: " + itemsToLoad +
"общим весом " + totalWeight + " кг.");
                        System.out.println(name + " отправляется на
разгрузку...");
                        Thread.sleep(2000);
                        System.out.println(name + " вернулся на склад.");
                    }

                    semaphore.release();

                    if (currentIndex >= items.length) {
                        break;
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
```

Захват семафора:

```
semaphore.acquire();
```

Грузчик захватывает семафор, что позволяет ему начать загрузку. Если все семафоры заняты, поток будет ожидать.

Инициализация переменных:

```
int totalWeight = 0;
StringBuilder itemsToLoad = new StringBuilder();
```

Инициализируются переменные для хранения общего веса и строки с весами загруженных предметов.

Цикл загрузки предметов:

```
                    while (true) {
                        int itemWeight;

                        synchronized (lock) {
                            if (currentIndex >= items.length) {
                                break;
                            }
                            itemWeight = items[currentIndex];
                            currentIndex++;
                        }

                        if (totalWeight + itemWeight <= MAX_WEIGHT) {
                            totalWeight += itemWeight;
                            itemsToLoad.append(itemWeight).append(" кг, ");
```

```
                    } else {

                        synchronized (lock) {
                            currentIndex--;
                        }
                        break;
                    }
                }
```

В этом цикле грузчик загружает предметы, пока общий вес не превысит MAX_WEIGHT. Если превышает, предмет возвращается обратно в массив.

Вывод сообщений и имитация разгрузки:

```
                if (totalWeight > 0) {
                    System.out.println(name + " загрузил: " + itemsToLoad +
"общим весом " + totalWeight + " кг.");
                    System.out.println(name + " отправляется на
разгрузку...");
                    Thread.sleep(2000);
                    System.out.println(name + " вернулся на склад.");
                }
```

Если грузчик загрузил хотя бы один предмет, выводятся сообщения о загрузке и имитации разгрузки.

Освобождение семафора:

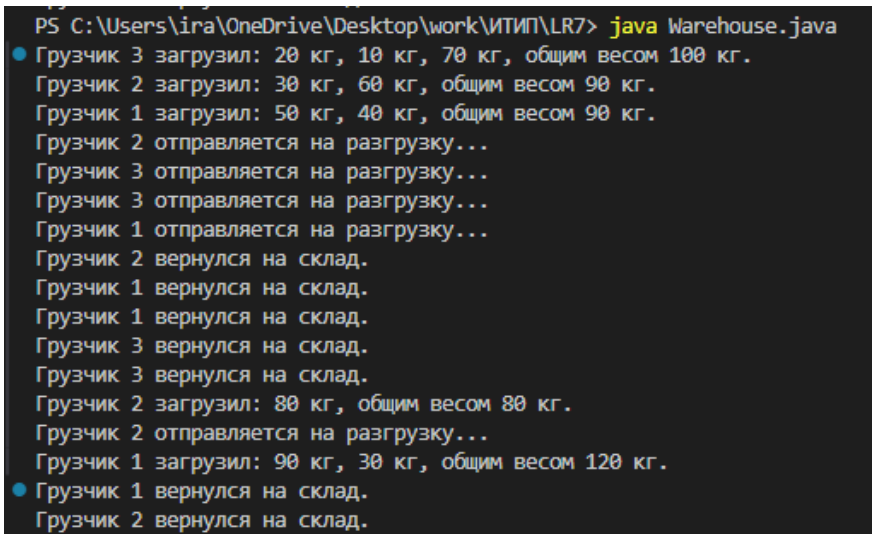
```
semaphore.release();
```

Грузчик освобождает семафор, позволяя другому грузчику начать загрузку.

Проверка завершения работы:

```
                if (currentIndex >= items.length) {
                    break;
                }
```

Если все предметы загружены, цикл завершается.



Вывод:

Корректно выполнены все задания с применением многопоточности в java.

Ссылка на репозиторий с кодом
https://github.com/k00kzaAntonovaIra/ITIP_2024.git