Домашнее задание №3. Многопоточность

Задание направлено на отработку навыков разработки многопоточных задач в концепции языка Java. Данная работа состоит из двух задач. Первая оценивающая в два балла и вторая в три. Для каждой задачи обязательна реализация модульных тестов, JavaDoc для нетривиальных случаев, и организации сборки с использованием системы Maven

Итого, за работу можно получить 5 баллов.

Задача №1. Корабли (2 балла)

Есть транспортные корабли, которые плывут от “генератора” к причалам для погрузки разного рода товаров. Они проходят через узкий туннель где одновременно могут находиться только 5 кораблей. Прохождение через туннель занимает некоторое время (пусть 1 секунда для каждого корабля. Так как одновременно в тоннеле могут находиться 5 кораблей, это означает, что при одновременном выпуске с генератора 5 кораблей, они пройдут через тоннель за одну секунду).

Каждый корабль представляет из себя отдельный поток java (Thread). Соответственно генератор кораблей генерирует объекты, которые реализуют интерфейс Runnable, это можно делать через ExecutorPool.

Существует 3 Типа кораблей (с хлебом, с бананами и с одеждой) и три вида вместительности 10, 50, 100 шт. товаров. 3 типа кораблей * 3 вида вместительности = 9 разных видов кораблей. Далее есть 3 вида причалов для погрузки кораблей — Хлеб, Банан и Одежда. Каждый причал берет или подзывает к себе необходимый ему корабль и начинает его загружать. За одну секунду причал загружает на корабль 10 ед. товара. То есть если у корабля вместительность 50 шт., то причал загрузит его за 5 секунд своей работы. Причал может загружать только по одному кораблю.

Требования к задаче:

- Правильно разбить задачу на параллельность.
- Синхронизировать потоки, сохранить целостность данных. Ведь ограничить доступ потоков к общему ресурсу дело не сложное, а заставить их работать согласованно уже намного сложнее. Проверку синхронизации потоков можно сделать инструментами логирования (по желанию можно использовать класс Logger из java.util, либо обычный system.out)
- Работа генератора кораблей не должна зависеть от работы причалов и наоборот.
- Общий ресурс должен быть Thread Safe
- Потоки не должны быть активными если нет задач.
- Потоки не должны держать mutex если нет задач.

Задача №2. TokenRing (3 балла)

Необходимо реализовать протокол TokenRing: узлы образуют топологию кольцо, данные могут передаваться по кольцу по часовой стрелке от 1-ко узла к n-му, после чего передача идёт от n-го к 1-му, если n-ый узел не является узлом назначения.

Каждый узел представляет из себя экземпляр класса Node, основной вид которого приведен ниже. Передаваемая информация также является отдельным классом DataPackage, который содержит время, когда информация создавалась (необходимо для учета статистики среднего времени доставки), узел назначения и рандомное строковое значение, которое и является "данными".

Данные в отдельном потоке "обрабатываются" на узле - то есть между моментами приема и дальнейшей передачи данных узел держит на себе пакет данных некоторое время, поток засыпает на 1 милисекунду.

Узел изначально хранит все данные в буфере, который должен быть потокобезопасным. То есть при поступлении данных на узел, они записываются в BufferStack. Каждый узел может обрабатывать ограниченное количество данных (например, 3) - если на узел поступает больше данных (например, 4), один пакет ждёт в буфере, пока узел не освободится для его приёма.
Количество узлов, количество данных на каждом узле, которые потом пойдут по кругу, а также файл для записи логов передаются параметрами в RingProcessor. Данные гоняются по кругу не бесконечно (иначе теряется смысл задачи). У каждого пакета данных есть узел-пункт назначения. Когда пакет данных достигает назначителя, последний записывает его на узел-координатор, который сохраняет его к себе в коллекцию. Координатор выбирается при инициализации кольца.
Весь процесс работы программы должен логгироваться. Логи должны содержать:

- Начало работы. Сколько узлов в топологии и количество данных на каждом узле. Номер координатора
- Фиксирование каждого пакета данных, откуда и куда он был передан. То есть должен записываться каждый акт передачи пакета данных
- В конце работы: фиксирование средней задержки в сети (то есть за сколько времени пакет с данными достигает узла назначения)
- Фиксирование средней задержки в буфере (то есть сколько времени в среднем узел находится в буфере) Логгирование можно производить стандартным инструментом из java.util.logging.Logger.

Работа программы завершается, когда последний пакет данных достигнул своего пункта назначения (соответствующего узла) и был сохранён на координаторе.

Примеры кода для реализации
Класс узла

```
public class Node extends Runnable {
    private final int nodeId;

    private final int corId;

    private BufferStack<DataPackage> bufferStack = new BufferStack<>();

    public List<DataPackage> allData;

    Node(int nodeId, corId) {
        this.nodeId = nodeId;

        this.coreId = coreId;

        if(nodeId == corId)
            allData = new ArrayList<>();
    }

    public long getId() {
    }

    public void setData(DataPackage dataPackage) {
    }

    public DataPackage getData() {
    }

    public BufferStack<DataPackage> getBuffer() {
    }
```

```
    /**
     * Начало работы узла. То есть из Node.bufferStack берётся пакет с данными
     * и отправляется на обработку, после чего передаётся следующему узлу.
     * Тут заключена логика, согласно которой обрабатываться может только 3 пакета данных
одновременно.
     */
    @Override
    public void run() {
    }
}
```

Класс данных, которые гоняются по кругу

```
public class DataPackage {
    private final int destinationNode;

    private final String data;

    private final long startTime;

    DataPackage(int destinationNode, String data) {
        this.destinationNode = destinationNode;

        this.data = data;

        // Фиксируется время, когда создаётся пакет данных. Необходимо для
        // вычисления времени доставки до узла назначения.
        startTime = System.nanoTime();
    }


    public int getDestinationNode() {
    }

    public long getStartTime() {
    }

    public String getData(){
        return data;
    }
}
```

Класс, ответственный за логику работы кольца

```
/**
 * В конструкторе кольцо инициализируется, то есть создаются все узлы и данные на узлах.
 * В методе {@link RingProcessor#startProcessing()} запускается работа кольца - данные
начинают
 * обрабатываться по часовой стрелке. Также производится логгирование в {@link
RingProcessor#logs}.
 * Вся работа должна быть потокобезопасной и с обработкой всех возможных исключений.
Если необходимо,
 * разрешается создавать собственные классы исключений.
 */
public class RingProcessor {

    private final int nodesAmount;
```

```
    private final int dataAmount;

    private final File logs;

    private List<Node> nodeList;

    private final Logger logger;

    /**
     * Сюда идёт запись времени прохода каждого пакета данных.
     * Используется в {@link RingProcessor#averageTime()} для подсчета среднего времени
     * прохода данных к координатору.
     */

    List<Long> timeList;

    RingProcessor(int nodesAmount, int dataAmount, File logs){
        this.nodesAmount = nodesAmount;

        this.dataAmount = dataAmount;

        this.logs = logs;

        logger = Logger.getLogger("ringLogger");

        init();
    }

    // Считается среднее время прохода.
    private long averageTime() {
        return 0;
    }

    private void init(){
        //initialize ring
    }

    public void startProcessing(){

    }
}
```

Точка входа в программу

```
public class Main {

    public static void main(String[] args) {
        RingProcessor processor = new RingProcessor(10, 3, new File("logPath"));

        processor.startProcessing();
    }
}
```