Практика №3. Многопоточность

Практика включает два задания по три балла каждое.

Задача №1. Пул потоков (3 балла)

В рамках данного задания необходимо разработать функциональный аналог пула потоков для управления процессом параллельного выполнения задач. Полученное решение должно предоставлять возможность:

- Создавать новый тред пул с указанным максимальным числом тредов
- Задавать тред пулу список задач для выполнения
- Управлять процессом завершения работы тред пула.

Представленное решение никоим образом не должно использовать готовые реализации тред пулов. Необходимо всячески минимизировать использование блокировок.

Тред пул будет включать две категории тредов:

- Быстрые (основной)
- Медленные (вспомогательные)

Быстрый тред всегда ровно один. Количество медленных тредов настраивается при создании тред пула.

Логика функционирования тред пула включает следующие действия:

- После добавления задачи она попадает в очередь и начинает исполняется быстрым тредом.
- Если время исполнения превышает заданное, т.е. задача оказывается “медленной”, то быстрый поток перестает её исполнять, а задача встает в очередь исполнения медленными потоками.
- Первый освободившийся медленный поток берет задачу на исполнение. Если время исполнения на медленном потоке превышает заданное, то происходит отказ выполнения задачи.

ДОПУСТИМ ДРУГОЙ ВАРИАНТ РЕШЕНИЯ. Вместо выполнения отказа на медленном тред пуле, допустима приостановка выполнения медленного задания с перестановкой его в конец очереди. Такие супер-медленные задачи должны всегда выполняться только при наличии свободных тредов, не занятых обычными задачами. Если выбран такой вариант решения, это необходимо обозначить в документации и соответствующим образом модифицировать последующие требования.

Результатом выполнения каждой задачи является такой объект, который позволяет:

- Получить результат, при этом заблокировав выполнение дальнейшего кода до его получения
- Добавить коллбэк для выполнения действия после получения результата
- Добавить коллбэк для выполнения действия при отказе в выполнении задачи
- Добавить коллбэк для выполнения действия при выбрасывании исключения во время выполнения задачи

При создании тред пула можно указать любую комбинацию следующих параметров (т.е. не все параметры обязаны быть заданы, и задавать их можно в любом порядке):

- Количество медленных тредов. Если не указано – то их число должно быть на один меньше числа ядер системы

- Предельное время выполнения задачи на быстром треде. Если не задано, то это 2 миллисекунды.
- Предельное время выполнения задачи на медленном треде. Если не задано, то это 0, и отказ не происходит никогда.

Разработанный класс также должен предоставлять следующие методы:

- execute - принимает задачу и ставит её в список на исполнение. Если тред пул уже завершил свою работу – бросается исключение.

- public void shutdownAfterAllTasks (){}

метод ожидает завершения ВСЕХ задач. Постановка новых задач в очередь после вызова этого метода невозможна

- public void shutdownAfterAllFastTasks (){}

метод ожидает завершения БЫСТРЫХ задач. Медленные задачи из очереди не выполняются, если какие-то выполнялись к моменту завершения всех быстрых задач – то по ним происходит отказ. Постановка новых задач в очередь после вызова этого метода невозможна

- public void shutdownNow (){}

все задачи находящиеся в процессе исполнения прерываются, очередь очищается, тред пул считается закрытым.

- public boolean isClosed(){}

проверяет, есть ли возможность отправить в пул новую задачу.

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 точки назначения, 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();
    }
}
```