Лабораторная работа № 4

Тема работы:

Синхронизация потоков. Проблема производителя и потребителя.

Теоретическая часть:

В многопоточных программах часто может возникать ситуация, когда несколько потоков пытаются получить доступ к одним и тем же ресурсам, что в итоге приводит к ошибочным и непредвиденным результатам.

Зачем нужна синхронизация в Java?

Синхронизация Java используется для того, чтобы с помощью некоторого метода синхронизации убедиться, что только один поток может получить доступ к ресурсу в данный момент времени.

Синхронизированные блоки Java

Java предоставляет возможность создания потоков и синхронизации их задач с помощью синхронизированных блоков. Синхронизированный блок в Java синхронизируется на некотором объекте. Все синхронизированные блоки синхронизируются на одном и том же объекте, и в них одновременно может выполняться только один поток. Все остальные потоки, пытающиеся войти в синхронизированный блок, блокируются до тех пор, пока поток, находящийся внутри синхронизированного блока, не выйдет из него.

```
synchronized(sync_object)
{
   // Access shared variables and other
   // shared resources
}
```

Переведено с помощью www.DeepL.com/Translator (бесплатная версия)

Источник: https://www.geeksforgeeks.org/synchronization-in-java/

Синхронизированные методы

Язык программирования Java предоставляет две основные идиомы синхронизации: синхронизированные методы и синхронизированные операторы. Более сложная из них - синхронизированные операторы - описана в следующем разделе. В данном разделе речь пойдет о синхронизированных методах. Чтобы сделать метод синхронизированным, достаточно добавить в его объявление ключевое слово synchronized:

```
public class SynchronizedCounter {

    private int c = 0;

    public synchronized void increment() {

        c++;

    }

    public synchronized void decrement() {

        c--;

    }

    public synchronized int value() {

        return c;

    }

}
```

Переведено с помощью www.DeepL.com/Translator (бесплатная версия)

Источник: https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Методы для синхронизации потоков (a.k.a монитор)

- **wait()** – заставляет текущий поток ждать, пока какой-либо другой поток не вызовет **notify()** или **notifyAll()** для того же объекта.
- **wait(long timeout)** – с помощью этого метода можно задать таймаут, по истечении которого поток будет автоматически разбужен. Поток может быть разбужен до истечения таймаута с помощью функций **notify()** или **notifyAll()**.
- **wait(long timeout, int nanos)** – это еще одна сигнатура, обеспечивающая ту же функциональность. Единственное отличие заключается в том, что мы можем обеспечить более высокую точность. Общее время ожидания (в наносекундах) вычисляется как 1_000_000*timeout + nanos.
- **notify()** – для всех потоков, ожидающих на мониторе данного объекта (с помощью любого из методов **wait()**), метод notify() уведомляет любой из них о необходимости произвольного пробуждения. Выбор того, какой именно поток будить, является недетерминированным и зависит от реализации. Поскольку метод notify() будит один случайный поток, мы можем использовать его для реализации взаимоисключающей блокировки, когда потоки выполняют схожие задачи. Однако в большинстве случаев целесообразнее реализовать **notifyAll()**.
- **notifyAll()** – этот метод просто пробуждает все потоки, ожидающие на мониторе этого объекта. Пробужденные потоки будут конкурировать обычным образом, как и любой другой поток, пытающийся синхронизироваться на этом объекте.

```
public class Data {

    private String packet;

    // True if receiver should wait

    // False if sender should wait
```

```
    private boolean transfer = true;
    public synchronized String receive() {
        while (transfer) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Thread Interrupted");
            }
        }
        transfer = true;
        String returnPacket = packet;
        notifyAll();
        return returnPacket;
    }
    public synchronized void send(String packet) {
        while (!transfer) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Thread Interrupted");
            }
        }
        transfer = false;
        this.packet = packet;
        notifyAll();
    }
}
```

Переведено с помощью www.DeepL.com/Translator (бесплатная версия)

Источник: https://www.baeldung.com/java-wait-notify

Проблема производителя и потребителя (также известна как **проблема ограниченного буфера**) заключается в синхронизации двух процессов использующий один буфер (хранилище) с информацией. Процессы производителей и потребителей

являются отдельными цикличными процедурами выполняющие соответствующие их назначениям последовательность действий. Эти процедуры обращаются к общему буферу для внесения или вынесения какой-либо информации и необходимо реализовать синхронизовать эти циклы друг с другом, таким образом, чтобы не происходило конфликтов при обращении к общему буферу (по типу – запрашиваемая информация отсутствует или пространство буфера уже заполнено).

Основное задание:

Основная задача состоит в создании синхронизации между производителем и потребителем. Когда производитель производит некий товар, то следственно потребитель должен его потребить. И, следовательно, если товаров нет на складе, то и потребитель не может приобрести товар. Реализовать алгоритм, в котором будут исполняться процесс производителя, генерирующий какие-либо товары и заносит их в условный склад и потребителя (может быть пользователь, либо выполняться автоматически вместе с производителями) который приобретает те или иные товары.

- **Легкий вариант:** реализовать алгоритм, в котором существует один или несколько производителей и один потребитель, которого будет изображать пользователь через консоль терминала. По алгоритму производитель производит продукт, далее по запросу пользователя показывается список с произведёнными продуктами, пользователь должен ввести название продукта для потребления.
- **Средний вариант:** реализовать алгоритм, в котором существует один или несколько производителей и несколько потребителей. Производитель производит товар и потребители потребляют товар. Сам процесс должен происходить автоматически (без вмешательства пользователя) и потребление товара, как и его производство должно происходить случайным образом.
- **Сложный вариант:** реализовать алгоритм, в котором существует один или несколько производителей и несколько потребителей. Роль одного из потребителей будет исполнять пользователь, остальные потребители исполняются алгоритмом программы. Процесс производства производителем и потребления потребителем, должен происходить автоматически, поверх запроса пользователя-потребителя. Потребление и производство также производится случайным образом, кроме потребителя пользователя терминалом.

Варианты для реализации задания:

- Хлебобулочные изделия и молочные продукты.
- Медикаменты и средства индивидуального ухода.
- Строительные материалы и средства индивидуальной защиты.
- Программное обеспечение и мультимедиа.
- Одежда и предметы обихода.

- Электроника и аксессуары.

По завершению работы, составьте отчет, в котором должно быть – Ваша фамилия, имя, группа, тема работы, Ваш вариант для реализации задания, краткое описание реализации задания, ссылку на исходный код на GitHub. Исходный код push-ите в вашу ветку в соответствующем репозитории - https://github.com/FCIM-PCD/Practice-Work-RU. Сохранить отчет в формате PDF и отправить на ELSE - https://else.fcim.utm.md/mod/assign/view.php?id=5828.