Лабораторная работа № 5

Тема работы:

Использование барьеров и семафоров.

Теоретическая часть:

CyclicBarrier

Средство синхронизации, позволяющее набору потоков ждать друг друга до достижения общей точки барьера. Циклические барьеры полезны в программах с фиксированным количеством потоков, которые должны периодически ждать друг друга. Барьер называется циклическим, потому что он может быть использован повторно после освобождения ожидающих потоков.

Конструкторы:

- **`CyclicBarrier(int parties)`** – Создает новый CyclicBarrier, который отключается, когда заданное количество потоков (**parties**) ожидают его, и не выполняет предопределенного действия, когда барьер отключается.
- **`CyclicBarrier(int parties, Runnable barrierAction)`** – Создает новый циклический барьер CyclicBarrier, который отключится, когда заданное количество потоков будут ожидать его, и который выполнит заданное действие барьера, когда барьер отключится, выполняемое последним потоком, вошедшим в барьер.

Методы:

- **`await()`** – ожидает, пока все потоки не вызовут await на этом барьере.
- **`await(long timeout, TimeUnit unit)`** – ожидает, пока все потоки не вызовут await на этом барьере или пока не истечет указанное время ожидания.
- **`getNumberWaiting()`** – возвращает количество потоков, ожидающих в данный момент у барьера.
- **`getParties()`** – возвращает количество потоков, необходимых для преодоления этого барьера.
- **`isBroken()`** – спрашивает, находится ли данный барьер в сломанном состоянии.
- **`reset()`** - сбрасывает барьер в исходное состояние.

```
public class TechLead extends Thread {
    CyclicBarrier cyclicBarrier;
    public TechLead (CyclicBarrier cyclicBarrier, String name) {
        super(name);
        this.cyclicBarrier = cyclicBarrier;
    }
    public void run() {
```

```
        try {
            Thread.sleep(3000);
            System.out.println(
                Thread.currentThread().getName() + " recruited developer");
            System.out.println(
                Thread.currentThread().getName() + " waiting to complete ...");
            cyclicBarrier.await();
            System.out.println("All finished recruting, " +
                Thread.currentThread().getName() +
                " gives offer letter to candidate");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
public class HRManager {
    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(3);
        TechLead techLead1 = new TechLead(cyclicBarrier,"John TL");
        TechLead techLead2 = new TechLead(cyclicBarrier,"Doe TL");
        TechLead techLead3 = new TechLead(cyclicBarrier,"Mark TL");
        techLead1.start();
        techLead2.start();
        techLead3.start();
        System.out.println("No work");
    }
}
```

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

Источник: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html

Semaphore

Счетный семафор. Концептуально, семафор хранит набор разрешений. Каждая функция **acquire()** при необходимости блокируется, пока не появится разрешение, а затем берет его. Каждая функция **release()** добавляет разрешение, потенциально освобождая блокирующий приобретатель. Однако никаких реальных объектов разрешений не используется; семафор просто ведет подсчет количества доступных разрешений и действует соответствующим образом. Семафоры часто используются для ограничения числа потоков, которые могут получить доступ к некоторому (физическому или логическому) ресурсу. Например, вот класс, который использует семафор для контроля доступа к пулу элементов:

```
class Pool {
    private static final int MAX_AVAILABLE = 100;
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);
    public Object getItem() throws InterruptedException {
        available.acquire();
        return getNextAvailableItem();
    }
    public void putItem(Object x) {
        if (markAsUnused(x))
            available.release();
```

```
    }
    // Not a particularly efficient data structure; just for demo
    protected Object[] items = ... whatever kinds of items being managed
    protected boolean[] used = new boolean[MAX_AVAILABLE];
    protected synchronized Object getNextAvailableItem() {
        for (int i = 0; i < MAX_AVAILABLE; ++i) {
            if (!used[i]) {
                used[i] = true;
                return items[i];
            }
        }
        return null; // not reached
    }
    protected synchronized boolean markAsUnused(Object item) {
        for (int i = 0; i < MAX_AVAILABLE; ++i) {
            if (item == items[i]) {
                if (used[i]) {
                    used[i] = false;
                    return true;
                } else
                return false;
            }
        }
        return false;
    }
}
```

Перед получением элемента каждый поток должен получить разрешение от семафора, гарантирующее, что элемент доступен для использования. Когда поток закончил работу с элементом, он возвращается обратно в пул, а разрешение возвращается на семафор, позволяя другому потоку получить этот элемент. Обратите внимание, что при вызове acquire() не происходит блокировки синхронизации, так как это помешало бы вернуть элемент в пул. Семафор инкапсулирует синхронизацию, необходимую для ограничения доступа к пулу, отдельно от синхронизации, необходимой для поддержания согласованности самого пула.

Семафор, инициализированный единицей и используемый таким образом, что для него доступно не более одного разрешения, может служить в качестве блокировки взаимного исключения (**`new Semaphore(1)`**). Такой семафор чаще называют бинарным, поскольку он имеет только два состояния: одно доступное разрешение или ноль доступных разрешений. При таком использовании бинарный семафор обладает тем свойством (в отличие от многих реализаций **Lock**), что "замок" может быть освобожден потоком, отличным от владельца (поскольку семафоры не имеют понятия собственности). Это может быть полезно в некоторых специальных контекстах, например, при восстановлении тупиковых ситуаций.

Как правило, семафоры, используемые для контроля доступа к ресурсам, должны инициализироваться как справедливые (**`new Semaphore(…, true)`**), чтобы гарантировать, что ни один поток не лишится доступа к ресурсу. При использовании семафоров для других видов управления синхронизацией преимущества несправедливого упорядочивания по пропускной способности часто перевешивают соображения справедливости.

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

Источник: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html

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 используя барьеры.
- **Средний вариант:** реализовать алгоритм используя барьеры и семафоры.
- **Сложный вариант:** реализовать алгоритм используя барьеры и семафоры, также имплементировать одного потребителя через управление пользователем.

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=8242.