HSLU Hochschule Luzern

Algorithmen & Datenstrukturen (AD)

Übung: Thread Steuerung (N2)

Themen: Warten auf Bedingungen und die Umsetzung mit **`wait`**, **`notify`** und **`notifyAll`**, Thread Steuerung durch Semaphore,
Zeitbedarf: ca. 240min.

Roger Diehl, Version 1.0 (HS 2024)

1 Wait-Pool-Demo (ca. 30')

1.1 Lernziele

- Code-Beispiel mit mit **`wait`**, **`notify`** und **`notifyAll`** nachvollziehen, Fehler finden und modifizieren.

1.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N21 – Abschnitt „Warten auf Bedingungen“. Zudem erhalten Sie die Code Vorlage `ch.hslu.ad.exercise.n2.waitpool`.

1.3 Aufgabe

Gegeben ist die Klasse `MyTask` …

```
/**
 * Einfacher Task für die Demonstration eines Wait-Pools.
 */
public final class MyTask implements Runnable {

    private static final Logger LOG =
        LogManager.getLogger(ch.hslu.ad.exercise.MyTask.class);
    private final Object lock;

    public MyTask(final Object lock) {
        this.lock = lock;
    }

    @Override
    public void run() {
        LOG.info("warten...");
        synchronized (lock) {
            try {
                wait();
            } catch (InterruptedException ex) {
                return;
            }
        }
        LOG.info("...aufgewacht");
    }
}
```

…und die Klasse `DemoWaitPool`.

```java
/**
 * Demonstration eines Wait-Pools.
 */
public final class DemoWaitPool {

    private static final Object LOCK = new Object();

    public static void main(final String args[]) throws InterruptedException {
        final MyTask waiter = new MyTask(LOCK);
        new Thread(waiter).start();
        Thread.sleep(1000);
        LOCK.notify();
    }
}
```

a) Es soll der Wait-Pool eines Objektes demonstriert werden. Die Klassen kompilieren problemlos. Trotzdem sind sie semantisch nicht in Ordnung.

- Welche Anmerkungen oder Bugs macht/findet die IDE zu `DemoWaitPool`?
- Welche Anmerkungen oder Bugs macht/findet die IDE zu `MyTask`?
- Versuchen Sie die Anmerkungen und Verbesserungsvorschläge nachzuvollziehen.

b) Führen Sie das Programm `DemoWaitPool` ohne Korrekturen aus.

- Was passiert bei der Ausführung von `DemoWaitPool`?
- Wie erklären Sie sich das Verhalten der Klassen?
- Welche minimalen Korrekturen sind nötig?
- Gibt es noch andere Korrektur-Varianten?

c) Sie haben nach der Korrektur in der main-Methode einen `synchronized` Block. Verschieben Sie die «`Thread.sleep`» Anweisung auch in den `synchronized` Block.

- Was passiert nun bei der Ausführung von `DemoWaitPool`?
- Wie erklären Sie sich das Verhalten?

1.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Was ist bei der Benachrichtigung mit Hilfe der **`notify`**/**`notifyAll`** Methoden zu beachten?
- Warum wird für die Benachrichtigung **`notifyAll`** empfohlen, statt **`notify`**?
- Was ist zu berücksichtigen, wenn man für die Benachrichtigung **`notifyAll`** verwendet?

2 Pferderennen (ca. 60')

2.1 Lernziele

- Code-Beispiel mit mit **`wait`**, **`notify`** und **`notifyAll`** nachvollziehen und modifizieren.

2.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N21 – Abschnitt „Warten auf Bedingungen“ und den dazugehörigen Regeln. Zudem erhalten Sie die Code Vorlage **`ch.hslu.ad.exercise.n2.latch`**.

2.3 Aufgabe

Ein praktischer Synchronisationsmechanismus ist das Latch. Latches sperren so lange, bis sie einmal ausgelöst werden. Danach sind sie frei passierbar. Sie sollen nun ein Latch in Java schreiben. Dazu soll das Interface `Synch` verwendet, bzw. implementiert werden.

```
/**
 * Schnittstelle für die Zutrittsverwaltung geschützter Bereiche.
 */
public interface Synch {

    /**
     * Eintritt in einen geschützten Bereich erlangen,
     * falls kein Zutritt möglich ist warten.
     * @throws InterruptedException wenn das Warten unterbrochen wird.
     */
    public void acquire() throws InterruptedException;

    /**
     * Freigabe des geschützten Bereiches.
     */
    public void release();
}
```

Zur Demo veranstalten wir ein kleines Pferderennen. Dazu gibt es Pferde...

```
/**
 * Ein Rennpferd, das durch ein Startsignal losläuft.
 * Nach einer zufälligen Zeit kommt es im Ziel an.
 */
public final class RaceHorse implements Runnable {

    private static final Logger LOG; //...ist zu initialisieren
    private final Synch startSignal;
    private final String name;
    private final Random random;
    /**
     * Erzeugt ein Rennpferd, das in die Starterbox eintritt.
     */
    public RaceHorse(final Synch startSignal, final String name) {
        this.startSignal = startSignal;
        this.name = name;
        this.random = new Random();
    }
    //...
```

```
    @Override
    public void run() {
        LOG.info("Rennpferd {} geht in die Box.", name);
        try {
            startSignal.acquire();
            LOG.info("Rennpferd {} laeuft los...", name);
            Thread.sleep(random.nextInt(3000));
        } catch (InterruptedException ex) {
            LOG.debug(ex);
        }
        LOG.info("Rennpferd {} ist im Ziel.", name);
    }
}
```

...und eine Rennbahn. Damit alle Pferde gerecht gestartet werden, kommen diese in eine Starterbox. Sobald diese geöffnet wird, sollen die Pferde loslaufen.

```
public final class Turf {

    private static final Logger LOG; //...ist zu initialisieren

    public static void main(final String[] args) {
        Synch starterBox = new Latch();
        for (int i = 1; i <= 5; i++) {
            Thread.startVirtualThread(new RaceHorse(starterBox, "Horse " + i));
        }
        LOG.info("Start...");
        starterBox.release();
    }
}
```

a) Implementieren Sie anhand der Vorgaben die Klasse `Latch`. Funktioniert die Applikation wie gewollt? Wenn nicht, verbessern Sie die Applikation.

b) Suchen Sie in der `java.util.concurrent` Bibliothek nach einem gleichen oder ähnlichem Synchronisationskonzept wie das `Latch` und verbessern Sie die Applikation Pferderennen.

c) **Optional**: Erweitern Sie die Applikation so, dass die Rennleitung einen Startabbruch (Interrupt) anordnen kann. Es genügt, wenn die Rennpferde (oder besser die Jockeys) merken, dass sie nicht ins Ziel laufen müssen und dies entsprechend Loggen.

d) **Optional**: Probieren Sie die neue Applikation aus, bei der die Rennleitung einen Startabbruch (Interrupt) durchführt.

2.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Ist das Rennen wirklich gerecht? Begründen Sie Ihre Antwort.
- Falls Ihre Antwort Nein ist – wie könnte man es gerechter machen?
- Was folgern Sie aus den obigen Überlegungen?

3 Signalgeber (ca. 60')

3.1 Lernziele

- Das Semaphor Konzept verstehen.
- Threads mit Hilfe eines Synchronisationsmechanismus beim Zugriff auf kritische Abschnitte steuern.

3.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N21 – Abschnitt „Verlorene Signale“. Zudem erhalten Sie die Code Vorlage `ch.hslu.ad.exercise.n2.signal`.

3.3 Aufgabe

Im Input N21 ist ein einfaches, nach oben nicht begrenztes, Semaphor vorgestellt worden. Diese rudimentäre Umsetzung soll als Basis dienen für ein paar Gedanken und eine umfangreichere Umsetzung dienen.

Sie müssen noch keinen Code schreiben.

a) Beantworten Sie folgende Fragen:
   1) Wie fair ist das im Input N21vorgestellte Semaphor?
   2) Was ist die Ursache für die entsprechende Fairness?
   3) Wie könnten Sie die bestehende Fairness verbessern?

b) Das im AD Input N21 (`N21_IP_ThreadSteuerung`) vorgestellte Semaphor hat in der Methode `release` noch Potential zur Verbesserung.
   1) Welche ist das?
   2) Was benötigen Sie um das Verbesserungspotential umzusetzen?

Ab diesem Punkt müssen Sie nun Code schreiben.

c) In C# ist das Semaphor nach oben begrenzt. Die MSDN definiert dazu den Konstruktor wie folgt (deutsche Übersetzung):

`Semaphore(Int32, Int32)`	Initialisiert eine neue Instanz der Semaphore-Klasse und gibt die ursprüngliche Anzahl von Einträgen und die maximale Anzahl von gleichzeitigen Einträgen an.

Erweitern Sie das im Input vorgestellte Semaphor, damit es nach oben begrenzt ist. Der Konstruktor Header könnte wie folgt aussehen:

```
/**
 * Erzeugt ein nach oben begrenztes Semaphor.
 *
 * @param permits Anzahl Passiersignale zur Initialisierung.
 * @param limit maximale Anzahl der Passiersignale.
 * @throws IllegalArgumentException wenn Argumente ungültige
 * Werte besitzen.
 */
public Semaphore(final int permits, final int limit)
        throws IllegalArgumentException
```

Machen Sie sich zu folgenden Punkten Gedanken:

1) Was sind ungültige Werte Argumente beim Konstruktor, d.h. wann wirft der Konstruktor eine `IllegalArgumentException`?
2) Wie initialisiert ein Default-Konstruktor die Attribute des nach oben begrenzten Semaphors?
3) Welche Methoden sind vom Limit des Semaphors betroffen?
4) Wie reagieren diese Methoden, wenn das Limit überschritten wird?

d) Erweitern Sie Ihr Semaphor noch mit folgenden Methoden, bei denen man mehr als ein Zutritt anfordern oder freigeben kann:
   - `void acquire(final int permits)`
   - `void release(final int permits)`

e) Probieren Sie Ihr erweitertes `Semaphor` aus. Prüfen Sie es vor allem auf ungültige Argumente.

3.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Wie würden Sie Ihr Semaphor einordnen – Windhund Prinzip Ja oder Nein? Begründen sie Ihre Antwort.
- Die Fragen aus a) betreffen nur das Semaphor. Wie würde im Allgemeinen eine faire Umsetzung beim „Warten auf Bedingungen“ aussehen.

4 Bounded Buffer (ca. 60')

4.1 Lernziele

- Das Semaphor Konzept anwenden.
- Threads mit Hilfe eines Synchronisationsmechanismus beim Zugriff auf gemeinsame Ressourcen steuern.

4.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N21 – Abschnitt „Bounded Buffer mit Semaphoren“. Zudem erhalten Sie die Code Vorlage `ch.hslu.ad.exercise.n2.buffer`.

4.3 Aufgabe

Wir wollen den Bounded Buffer mit Semaphoren auf dem Input erweitern. Als Code Vorlage ist die Klasse `BoundedBuffer` und eine Demo mit Produzenten und Konsumenten gegeben. Es gelten die folgenden Randbedingungen:

- Der Bounded Buffer wird in einer Umgebung eingesetzt, in welcher Produzenten und Konsumenten eine gemeinsame Datenstruktur zum Austausch von Elementen verwenden.
- Falls der Konsument Daten aus einem leeren Puffer auslesen will, wird er blockiert.
- Falls der Produzent Daten in einen vollen Puffer schreiben will, wird er blockiert.
- Der Bounded Buffer besitzt einen internen Buffer vom Datentyp `ArrayDeque`.
- Der Bounded Buffer hat eine fixe Grösse.
- Der Bounded Buffer benutzt die Semaphoren aus dem `java.util.concurrent` Package.

Lösen Sie die folgenden Teilaufgaben.

a) Ergänzen Sie je eine zusätzliche Methode `add` und `remove` mit Timeout.

b) Fügen Sie die folgenden Methoden hinzu:
   - `empty` – dient der Abfrage ob der Buffer leer ist
   - `full` – dient der Abfrage ob der Buffer voll ist
   - `size` – gibt die Anzahl Elemente im Buffer zurück

c) Probieren Sie Ihren `BoundedBuffer` mit der Produzenten-Konsumenten Demo aus.

4.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Warum müssen beim Bounded Buffer mit Semaphoren nicht mehr die ganzen Methoden synchronisiert sein?
- Sie haben bei denjenigen Methoden wo eine `InterruptedException` auftreten kann, diese an den Aufrufer weitergegeben. Warum macht dies Sinn?
- Wie verhält sich Ihr `BoundedBuffer` beim Eintreffen eines Interrupts? Haben Sie das ausprobiert?