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 (FS 2017)

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“.

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`

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

    private static final Object lock = new Object();

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

a) Man möchte damit den Wait-Pool eines Objektes mit demonstrieren. Die Klassen kompilieren problemlos. Trotzdem sind sie semantisch nicht in Ordnung.

- 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?

b) Es handelt sich bei dieser Übung nur um eine Demonstration. Trotzdem, lassen Sie Ihre Entwicklungsumgebung (SDK) den in a) korrigierten Code analysieren.

- Welche Anmerkungen oder Bugs macht/findet die SDK zu `DemoWaitPool`?
- Welche Anmerkungen oder Bugs macht/findet die SDK zu `MyTask`?
- Versuchen Sie die Anmerkungen und Verbesserungsvorschläge nachzuvollziehen. Warum kann man in diesem Code nicht alle Verbesserungsvorschläge umsetzen?

c) Fassen Sie den gesamten Code der main-Methode in einen `synchronized` Block zusammen. Nehmen Sie als Wait-Pool das `lock` Objekt.

- Was passiert 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`**?
- Wenn ein Thread einen andern Thread steuern will, ist dies offensichtlich keine gute Lösung. Wie sieht eine bessere Lösung aus?

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.

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 ein Interface `Synch` verwendet, bzw. implementiert werden.

```java
/**
 * 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 beim Austritt.
     */
    public void release();
}
```

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

```java
/**
 * 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 noch zu initialisieren
    private final Synch startSignal;
    private volatile Thread runThread;
    private final Random random;

    /**
     * Erzeugt ein Rennpferd, das in die Starterbox eintritt.
     * @param startSignal Starterbox.
     */
    public RaceHorse(Synch startSignal) {
        this.startSignal = startSignal;
        this.random = new Random();
    }

    //...
```

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

...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 noch zu initialisieren

    public static void main(final String[] args) {
        Synch starterBox = new Latch();
        for (int i = 1; i < 6; i++) {
            new Thread(new RaceHorse(starterBox),"Horse "+i).start();
        }
        LOG.info("Start...");
        starterBox.release();
    }
}
```

a) Implementieren Sie anhand der Vorgaben die Klasse `Latch`.

b) 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.

c) Erweitern Sie die Klasse `Latch` mit einem Timeout.

d) Erweitern Sie die Applikation so, dass bei einem verzögerten Rennstart das Timeout des `Latch` wirkt.

e) Testen Sie die neue Applikation, wo 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 Bounded Buffer (ca. 90')

3.1 Lernziele

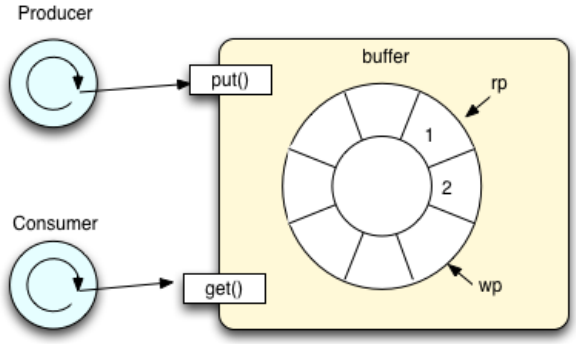
- Das Konzept der **„guarded blocks“** verstehen.
- Threads mit Hilfe eines Synchronisationsmechanismus beim Zugriff auf gemeinsame Ressourcen steuern.

3.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N21 – Abschnitt „Bounded Buffer“.

3.3 Aufgabe

Im Input N21 ist eine einfache Bounded-FIFO-Queue (ohne Semaphor Einsatz) vorgestellt worden. Diese rudimentäre Umsetzung soll als Basis dienen für einen umfangreicheren Bounded Buffer.



Randbedingungen: Der Bounded Buffer wird in einer Umgebung eingesetzt, in welcher Produzenten und Konsumenten eine gemeinsame Datenstruktur zur Austausch von Elementen verwenden. Der Bounded Buffer hat eine fixe Grösse.
Für einen Konsumenten ist es wichtig, dass er nicht Daten aus einem leeren Buffer auslesen kann. Der Produzenten auf der anderen Seite darf nicht in einen vollen Buffer hineinschreiben.

a) Erstellen Sie eine Klasse `BoundedBuffer`, in der man Elemente generischen Datentyps speichern kann. Sie können dazu die Methoden `put` und `get`, welche nach dem FIFO Prinzip funktionieren, aus dem Input N21 entnehmen. Nehmen Sie auch die `put` Methode mit Timeout in Ihre Klasse auf. Zusätzlich fügen Sie noch folgende 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

b) Testen Sie Ihren `BoundedBuffer` mit einer Test-Suite nach dem Produzenten / Konsumenten Prinzip.

c) Ergänzen Sie Ihren `BoundedBuffer` mit einer `get` Methode mit Timeout.

d) Testen Sie Ihren ergänzten `BoundedBuffer`.

e) Damit der Bounded Buffer in einer komplexeren Anwendung eingesetzt werden kann, müssen Sie Ihre Klasse `BoundedBuffer` mit zusätzlichen Methoden erweitern:
   - `front` – liefert das erste Element im Buffer zurück
   - `back` – liefert das letzte Element im Buffer zurück
   - `push` – fügt ein neues Element nach dem LIFO Prinzip in den Buffer ein
   - `pop` – entfernt ein Element nach dem LIFO Prinzip aus dem Buffer

f) Testen Sie den Bounded Buffer mit einer Produzenten / Konsumenten Applikation.

3.4 Reflektion

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

- Warum ist es gerade im Fall der Klasse `BoundedBuffer` nicht gut, wenn man als Lock- und Wait-Pool das aktuelle Objekt nimmt?
- Warum ist es im Fall der `BoundedBuffer` Klasse nicht schlimm, wenn man das aktuelle Objekt als Lock- und Wait-Pool genommen hat? Allerdings gilt dies nur, wenn man sich an die Regeln des „Warten auf Bedingungen“ gehalten hat.
- Sie haben bei den Methoden wo eine `InterruptedException` auftreten kann, diese an den Aufrufer weitergegeben. Warum haben Sie das getan?
- Wie verhält sich Ihr `BoundedBuffer` beim Eintreffen eines Interrupts? Haben Sie das getestet?
- Wieso macht es gerade beim `BoundedBuffer` Sinn `notifyAll` und nicht nur `notify` zu verwenden?
- Wieso macht es Sinn `notifyAll` nur aufgrund einer Bedingung aufzurufen?

4 Signalgeber (ca. 60')

4.1 Lernziele

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

4.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N21 – Abschnitt „Verlorene Signale“.

4.3 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? Es kein Code zu schreiben.

b) Das 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(int permits)`
   - `void release(int permits)`

e) Testen Sie Ihr `Semaphor`.

4.4 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.
- Wie viele Synchronisationsmechanismen beinhaltet der Bounded Buffer mit Semaphore? Welches sind diese? Können Sie sich einen Bounded Buffer mit weniger Synchronisationsmechanismen vorstellen?
- Was ist das grundsätzliche Problem in nebenläufigen Anwendungen mit Semaphoren und dadurch auch mit Bounded Buffer und ähnlichen Konstrukten?

5 Optional: Scatter/Gather-Verarbeitung

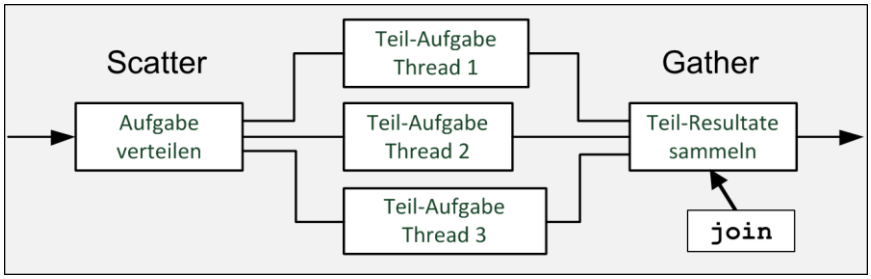
5.1 Lernziele

- Threads mit Hilfe eines Synchronisationsmechanismus beim Zugriff auf kritische Abschnitte oder gemeinsame Ressourcen steuern.
- Einsatz von Thread Pools.

5.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N21 und den vorher gegangenen Aufgaben.

Um Multicore-Systeme optimal zu nutzen, sollte eine Aufgabe aufgeteilt und diese Teile unabhängig voneinander von einzelnen Threads bearbeitet werden.



Die Scatter/gather-Verarbeitung ist ein Verfahren zur Berechnung eines komplexen Problems in mehreren Schritten. In der scatter-Phase werden dabei Teilprobleme an einzelne Prozesse (workers) zugeteilt, die daraufhin unabhängig voneinander Zwischenlösungen berechnen. In der gather-Phase werden diese Zwischenlösungen eingesammelt und zu einer Gesamt-Zwischenlösung zusammengefasst, die dann in einem weiteren scatter/gather-Zyklus erneut in Teilprobleme zerlegt wird. Dies wird solange wiederholt, bis das Endergebnis vorliegt.

5.3 Aufgabe

Sie sollen mit Hilfe eines Synchronisationsmechanismus eine `Scatter/gather`-Verarbeitung durchführen. Dazu erstellen Sie eine kleine Simulation, welche folgende Komponenten verwendet:

- Einen `Operator`, der alles steuert.
- Mehrere `Worker`s, die je eine Aufgabe bearbeiten.
- Der `Task`, die eigentliche Aufgabe.

Eine definierte Anzahl `Worker` holen beim `Operator` je eine Aufgabe ab. Bevor die Aufgaben zur Abholung bereit stehen muss der `Operator` diese initialisieren. Ist die Initialisierung abgeschlossen, wird die definierte Anzahl wartender `Worker` vom `Operator` befreit (released). Ein `Worker` holt sich dann eine Aufgabe und führt diese aus. Ist die Aufgabe beendet, wird der `Operator` vom `Worker` benachrichtigt. Sobald alle ausstehenden Aufgaben gelöst sind, initialisiert der `Operator` eine neue Anzahl Aufgaben.

Implementieren Sie diese kleine Simulation. Nutzen Sie dazu alle Konzepte aus dem Input N21. Machen Sie die Aufgaben nicht zu kompliziert, ein einfacher `Thread.sleep` genügt. Es geht um das Scatter/gather-Prinzip nicht um eine komplizierte Aufgaben Bewältigung.