Übung N2: Thread-Steuerung

Patrick Bucher

17.05.2017

Inhaltsverzeichnis

1 Wait-Pool-Demo

a)

Was passiert bei der Ausführung von `DemoWaitPool`?

Es kommt zu zwei Exceptions. Die eine wird vom `wait()`-Aufruf innerhalb der `run()`-Methode der Klasse `MyTask` geworfen, die andere vom `notify()`-Aufruf innerhalb der `main()`-Methode der Klasse `DemoWaitPool`.

Wie erklären Sie sich das Verhalten der Klassen?

Der Aufruf von `wait()` versucht `this` zu sperren: ein Objekt, das nicht gesperrt ist. Das hat eine `IllegalMonitorStateException` zur Folge. Da dies eine unchecked-Exception ist, wird die `run()`-Methode verlassen, ohne dass der Lock freigegeben wird.

Mit dem `notify()`-Aufruf auf das Lock-Objekt soll dann der Thread wieder geweckt werden. Doch `lock` wurde zuvor nicht gesperrt, was eine weitere `IllegalMonitorStateException`zur Folge hat.

Welche *minimalen* Korrekturen sind nötig?

In `MyTask` müsste `wait()` auf das Objekt `lock` ausgeführt werden:

```
synchronized (lock) {
    try {
        lock.wait();
    } catch (InterruptedException ex) {
        return;
    }
}
```

In `DemoWaitPool` müsste `lock` in einem `synchronized`-Block stehen:

```
synchronized (lock) {
    lock.notify();
}
```

Gibt es noch andere Korrektur-Varianten?

Statt auf das `lock`-Objekt könnte man direkt auf die `MyTask`-Instanz sperren. Es ginge hier auch ohne spezielles `lock`-Objekt.

b)

Eclipse bietet diese Funktionalität leider nicht, weswegen ich diese Aufgabe vorerst nicht mache.

c)

Was passiert bei der Ausführung von `DemoWaitPool`?

`MyTask` wartet ewig.

Wie erklären Sie sich das Verhalten?

`MyTask` kann keine Sperre auf `lock` erstellen, da `lock` bereits von `DemoWaitPool` gesperrt wurde, und wartet deswegen ewig.

Reflektion

Frage: Was ist bei der Benachrichtigung mit Hilfe der `notify`/`notifyAll`-Methoden zu beachten?

Antwort: Es muss vorher mindestens ein Thread wartend sein. Der `wait()`-Aufruf muss *vor* dem `notify()`/`notifyAll()`-Aufruf erfolgen.

Frage: Warum wird für die Benachrichtigung `notifyAll` statt `notify` empfohlen?

Antwort: `notify` weckt nur einen schlafenden Thread auf, `notifyAll` ermöglicht es allen schlafenden Threads aufzuwachen. Wird eine Aktion abgeschlossen, nach der es mehreren Threads wieder möglich wird weiterzuarbeiten, sollte `notifyAll` ausgeführt werden, damit soviele Threads wie möglich wieder arbeiten können.

Frage: Wenn ein Thread einen anderen Thread steuern will, ist dies offensichtlich keine gute Lösung. Wie sieht eine bessere Lösung aus?

Antwort: Eine übergeordnete Logik soll sich um die Steuerung der beiden Threads kümmern.

2 Pferderennen

Reflektion

Frage: Ist das Rennen wirklich gerecht? Begründen Sie Ihre Antwort.

Antwort: Die Pferde-Threads werden der Reihenfolge nach gestartet. In dieser Startreihenfolge reihen sie sich in der Regel auch in die `wait()`-Warteschlange ein. Beim Aufruf von

`notifyAll()` werden zwar alle dieser Threads auf einmal geweckt, da aber mein Laptop und mein PC jeweils nur über vier Cores verfügen, können nicht alle Rennpferde wirklich parallel starten.

Frage: Falls Ihre Antwort Nein ist – wie könnte man es gerechter machen?

Antwort: Ein wirklich gerechtes Rennen könnte es nur dann geben, wenn es mindestens pro Pferd einen CPU-Core gäbe.

Frage: Was folgern Sie aus den obigen Überlegungen?

Antwort: Wirkliche Gleichzeitigkeit ist für unser Konzept mit prozedural-imperativen Programmiersprachen kaum vorstellbar. Selbst wenn man über viele CPU-Cores verfügt, heisst das noch lange nicht, dass wirklich alle parallelen Operationen gleichberechtigt ausgeführt werden.

3 Bounded Buffer

[Hinweis: Die Methoden `front()`, `back()`, `push()` und `pop()` habe ich nicht implementiert, da sie meiner Meinung nach der ursprünglichen Konzeption der Klasse als FIFO-Queue zuwiderlaufen.]

Reflektion

Frage: Warum ist es gerade im Fall der Klasse `BoundedBuffer` nicht gut, wenn man als Lock- und Wait-Pool das aktuelle Objekt nimmt?

Antwort: Auf diese Weise wird Performance verschenkt. Die Operationen `put()` und `get()` arbeiten am jeweils anderen Ende des Buffers und sollten, sofern der Buffer weder voll noch leer ist, auch gleichzeitig ausgeführt werden können.

Frage: 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.

Antwort: In den Grenzfällen (bei leerem und vollem Buffer) muss zuerst `put()` oder `get()` durchgeführt werden, bevor die gegenteilige Aktion ausgeführt werden kann.

Frage: Sie haben bei den Methoden, wo eine `InterruptedException` auftreten kann, diese an den Aufrufer weitergegeben. Warum haben Sie das getan?

(Habe ich eigentlich nicht... aber ich hätte es scheinbar tun sollen.) In meiner Implementierung logge ich diese Exceptions bloss nach `System.err` und behandle sie somit nicht. Man sollte diese aber weitergeben, damit der Client entscheiden kann, was er in diesem Fall tun möchte, z.B. eine Zeit lang warten.

Frage: Wie verhält sich Ihr `BoundedBuffer` beim Eintreffen eines Interrupts? Haben Sie das getestet?

Antwort: Meine Implementierung loggt zwar den Interrupt, lässt die Programmlogik aber dann einfach weiterlaufen. Mein `BoundedBuffer` ignoriert Interrupts.

Frage: Warum macht es gerade beim `BoundedBuffer` Sinn `notifyAll` und nicht `notify` zu verwenden?

Antwort: In einem Szenario mit `m` Produzenten und `n` Konsumenten können folgende Fälle auftreten:

1. Der Buffer ist leer, die Konsumenten müssen warten.
2. Der Buffer ist voll, die Produzenten müssen warten.

`notify()` weckt nur einen Thread auf. Weckte er im Fall 1 einen Konsumenten bzw. im Fall 2 einen Produzenten auf, brächte das überhaupt nichts, da dieser weiter warten müsste. Mit `nofiyAll()` werden auch diejenigen Threads aufgeweckt, die wirklich weiterarbeiten können.

Frage: Warum macht es Sinn `notifyAll()` nur aufgrund einer Bedingung aufzurufen?

Antwort: In meinem Fall sind es zwei Bedingungen: Die Konsumenten können wieder lesen; die Produzenten können wieder schreiben.

4 Signalgeber

a)

Frage: Wie fair ist das im Input N21 vorgestellte Semaphor?

Antwort: So fair, wie sich dessen Clients verhalten.

Frage: Was ist die Ursache für die entsprechende Fairness?

Antwort: Das Semaphor ist “dumm”: es vergibt seine Ressourcen einfach an denjenigen Aufrufer, der im richtigen Moment, d.h. wenn gerade Ressourcen frei sind, anfragt.

Frage: Wie könnten Sie die bestehende Fairness verbessern?

Antwort: Die Anfragen müssten mit einer Warteschlange verwaltet werden.

b)

Das vorgestellte Semaphor hat in der Methode `release` noch Potential zur Verbesserung.

Frage: Welche ist das?

Antwort: `notifyAll` müsste nur dann aufgerufen werden, wenn die Variable `sema` vor dem Aufruf den Wert `0` gehabt hätte. Ansonsten ist kein Thread am Warten.

Frage: Was benötigen Sie um das Verbesserungspotenzial umzusetzen?

Antwort: Eine einfache `if`-Abfrage:

```
if (sema == 0) {
    this.notifyAll();
}
sema++;
```

c)

Frage: Was sind ungeültige Argumente beim Konstruktor, d.h. wann wirft der Konstruktor eine `IllegalArgumentException`?

Antwort: Wenn `permits` grösser als `limit` oder einer der Werte negativ ist.

Frage: Wie initialisiert ein Default-Konstruktor die Attribute des nach oben begrenzten Semaphors?

Antwort: `permits = 0` und `limits = Integer.MAX_VALUE`.

Frage: Welche Methoden sind vom Limit des Semaphors betroffen?

Antwort: Nur die `release()`-Methoden.

Frage: Wie reagieren diese Methoden, wenn das Limit überschritten wird?

Antwort: Sie werfen eine `IllegalStateException`.

Reflektion

Frage: Wie würden Sie Ihr Semaphor einordnen – Windhund-Prinzip Ja oder Nein? Begründen Sie Ihre Antwort.

Antwort: Mein Semaphor arbeitet nicht nach dem Windhund-Prinzip, da er die Anfragen nicht in einer Queue einordnet.

Frage: Die Fragen aus a) betreffen nur das Semaphor. Wie würde im Allgemeinen eine faire Umsetzung beim “Waren auf Bedingungen” aussehen?

Antwort: Die Anfragen müssten in eine Warteschlange geschrieben und auch in dieser Reihenfolge wieder abgearbeitet werden.

Frage: Wie viele Synchronisationsmechanismen beinhaltet der Bounded Buffer mit Semaphore? Welches sind diese? Können Sie sich einen Bounded Buffer mit weniger Synchronisationsmechanismen vorstellen?

Antwort: Es müssten wohl zwei Semaphoren sein: je eines für Lese- und Schreibzugriff. Es wäre auch mit einem Semaphore zu realisieren, wobei nur abwechslungsweise gelesen oder geschrieben werden dürfte.

Frage: Was ist das grundsätzliche Problem in nebenläufigen Anwendungen mit Semaphoren und dadurch auch mit Bounded Buffer und ähnlichen Konstrukten?

Antwort: Ein Semaphor unterscheidet nur zwischen Zugriff und Nicht-Zugriff. In vielen Anwendungen gibt es jedoch verschiedene Arten des Zugriffs, etwa Lese- und Schreibzugriff im Bounded Buffer. Diese Zugriffe sollten möglichst ausgeglichen vergeben werden.

5 Optional: Scatter/Gather-Verarbeitung

Ich habe eine `SumOperator` geschrieben, die ein grosses und zufälliges `int`-Array (bzw. einen `SumTask`) von mehreren `PartialSumWorker`-Instanzen aufsummieren lässt. Dazu wird das Array zunächst in ungefähr gleichgrosse Teilarrays aufgespalten, und die Teilsummen der Worker werden am Schluss wieder durch den Operator aufsummiert. Der Testfall `SumScatterGatherTest` demonstriert die Funnktionalität mit verschiedenen Tests, die eine unterschiedliche Anzahl von Threads verwenden (von einem Thread bis zu einem Thread pro Array-Element).