Aufgabenblatt 3

Threads und Synchronisation

1 Simulation einer Container-Verladestation (Semaphore und Lock)

Es soll ein Programm zur Simulation eines Güterverkehrszentrums entwickelt werden, bei dem Container an mehreren Verladerampen auf LKWs umgeladen werden. Wenn ein LKW beladen wird, fährt er zu der Verladerampe, an dem die kleinste Schlange an wartenden LKWs steht. Bei zwei oder mehr freien Verladerampen, d.h. ohne Warteschlange, wählt der LKW zufällig seine nächste Verladerampe aus.

Sobald der LKW an der Reihe ist, wird ein Container umgeschlagen (d.h., die Zugmaschine wird mit dem Container zusammengeführt). In dieser Simulation kann davon ausgegangen werden, dass LKWs immer direkt beladen werden können, es also nie zu Engpässen an Containern kommt. Um die fürs Umschlagen benötigte Zeit zu simulieren, wird eine zufällige Zeit lang gewartet (zwischen 0 und 1 Sekunden). Danach fährt der LKW die Containerware zu seinem Zielort - dieser Vorgang soll wieder durch eine Zufallszeit (zwischen 5 und 10 Sekunden) simuliert werden – und reiht sich wieder an der Verladestation für den nächsten Warenumschlag ein. Beim Umschlag soll für jede Verladerampe ein Zähler erhöht werden um am Jahresende (bzw. am Ende der Simulation) einen Überblick über die jeweilige Auslastung zu bekommen.

Die Simulation soll nach einer vorgegebenen Zeit enden und die Threads durch den Aufruf der Methode `interrupt()` beendet werden.

1.1 Erstellen des Java-Programms

Schreiben Sie ein Java-Programm, mit einer beliebig konfigurierbaren Anzahl an Verladerampen und LKWs. Der Verladeprozess selbst soll mit Hilfe der Synchnonisationsklassen `Semaphore` bzw. `ReentrantLock` implementiert werden. Jeder LKW soll dabei als eigener Thread modelliert werden und am Anfang nach einer Zufallszeit (zwischen 0 und 3 Sekunden) starten. Am Ende der Simulation soll auf das Ende aller LKW-Threads gewartet werden und sämtliche Zähler der Verladerampen ausgegeben werden.

Testen Sie die Simulation mit 20 LKW-Threads und 5 bzw. 10 Verladerampen für verschiedene (sinnvollen) Gesamt-Simulationszeiten. Testen Sie ihr Programm mehrmals und suchen und behebend Sie auch Fehler, die manchmal, sporadisch auftreten.

1.2 Hinweise/Randbedingungen

- **Alle Zugriffe auf Objekte, die von mehreren Threads verändert werden können, müssen synchronisiert werden!**
- **Aktives Warten ist an keiner Stelle im Programm erlaubt!**
- Nutzen Sie für die notwendigen Synchronisationselemente `Semaphore` bzw. `ReentrantLock` **unbedingt** das Package `java.util.concurrent`

- Erzeugen Sie einen Thread je LKW und geben Sie jedem Thread bei der Erzeugung Informationen über alle Verladestationen mit (z.B. in Form einer Liste), damit er seine Auswahlentscheidung treffen kann.
- Mehrere LKWs dürfen nicht die gleiche Warteschlange als kürzeste erkennen. Wie lässt sich das verhindern?
- Zugriffe auf kritische Abschnitte müssen so kurz wie möglich gehalten werden und Anfang und Ende müssen immer in derselben Methode definiert sein.
- Das sichere Beenden von kritischen Abschnitten muss auch im Falle einer `Exception` sichergestellt sein.
- Auch wenn diese Aufgabe nicht exakt der Erzeuger-Verbraucher-Problemstellung entspricht, so können Sie doch vergleichen, wie die Semaphore-Lösung dafür aussieht. Wo gibt es dort Warteschlangen?

2 Schere, Stein, Papier (Java-Monitore und ConditionQueues)

Zwei Threads sollen das bekannte Spiel „Schere, Stein, Papier“ (auch *Schnick, Schnack, Schnuck*) gegeneinander spielen (siehe https://de.wikipedia.org/wiki/Schere,_Stein,_Papier). Da Threads keine Hände besitzen, muss jeder Thread in jeder Runde ein Spielobjekt (entweder ein Schere-, Stein oder Papierobjekt) auf einen virtuellen Tisch legen. Anschließend nimmt ein Schiedsrichter-Thread die Auswertung vor und leert den Tisch wieder. Insgesamt gibt es also vier Threads: main-Thread, „Spieler 1“, „Spieler 2“ und der Schiedsrichter-Thread. Es soll solange eine Runde nach der anderen gespielt werden, bis nach einer vorgegebenen Zeit alle Threads (außer `main`) mittels des Interrupt-Mechanismus beendet worden sind. Danach soll eine Gesamtauswertung (Gesamtanzahl gespielter Runden, Anzahl Unentschieden, Anzahl Gewinne Thread 0 / Thread 1) ausgegeben werden.

2.1 Erstellen des Java-Programms

Schreiben Sie ein Java-Programm, welches das o.g. Spiel mittels Java-Threads realisiert. Benutzen Sie die Synchronisationsmechanismen („Monitor“: Eintritt in einen Objekt-Monitor mittels `synchronized`) und die Methoden zur Threadsynchronisation (`wait()`, `notify()`, `notifyAll()`). Für die folgende Aufgabe können Sie den Aufwand gering halten, wenn alle Synchronisationsobjekte und –Methoden in einer Klasse gekapselt werden, beispielsweise in einem zentralen „Tisch“-Objekt. Beachten Sie bei der Implementierung auch die Tipps unten!

Testen Sie ihr Programm mehrmals und dokumentieren Sie die erfolgreiche Spieldurchführung durch geeignete Testausgaben. Suchen und behebend Sie auch Fehler, die manchmal, sporadisch auftreten.

2.2 Variante mit ConditionQueues

Erzeugen Sie eine weitere Programmversion, die für die Synchronisation ausschließlich die im JAVA-Package `java.util.concurrent.locks` zur Verfügung gestellten Mechanismen (`locks` und `conditions`) mit mehreren ConditionQueues benutzt. Was ist jeweils eine geeignete Bedingung für eine ConditionQueue? Welche(r) Thread(s) wartet/warten darauf?

Testen Sie ihr Programm mehrmals und dokumentieren Sie die erfolgreiche Spieldurchführung durch geeignete Testausgaben. Suchen und behebend Sie auch Fehler, die manchmal, sporadisch auftreten.

2.3 Hinweise/Randbedingungen

- **Alle Zugriffe auf Objekte, die von mehreren Threads verändert werden können, müssen synchronisiert werden!**
- **Aktives Warten ist an keiner Stelle im Programm erlaubt!**

- Führen Sie die Problemstellung auf das aus der Vorlesung bekannte Synchronisationsproblem „Erzeuger/Verbraucher“ zurück
- Zugriffe auf kritische Abschnitte müssen so kurz wie möglich gehalten werden und Anfang und Ende müssen immer in derselben Methode definiert sein.
- Das sichere Beenden von kritischen Abschnitten muss auch im Falle einer `Exception` sichergestellt sein.
- Für die Repräsentation der Spielobjekte (Schere, Stein, Papier) bietet sich das `enum`-Konstrukt in Java an, da `enum`-Konstanten Java-Objekte sind, aber auch über einen Index (Methode `ordinal()`) verfügen und daher für Array-Zugriffe verwendbar sind.
- Ihre Lösungen für die Aufgaben 2.1 und 2.2 sollten sich am besten nur **in einer Datei unterscheiden**! Bestenfalls können Sie dasselbe Projekt für beide Abgaben verwenden und müssen nur die Konfiguration ändern.
- Stellen Sie die Spielregeln bzgl. der Gewinnermittlung (siehe https://de.wikipedia.org/wiki/Schere,_Stein,_Papier#Grundregeln) als Matrix dar!