Nebenläufige Programmierung

Blatt 2

Name:	Bernhard FRITZ
Matrikelnummer:	1316136
Gruppe:	11

Gelöste Aufgaben

1)	42%	/	42%
2)	33%	/	33%
3)	25%	/	25%
=	100%	/	100%

Starten der Aufgaben

1) MainTest
2) Process

3.a) LazyInitRace

3.b) ThreadSafeInit1, ThreadSafeInit2

Aufgabe 1

Damit ein Consumer nur dann konsumiert, wenn alle Producer eine Zahl bereitstellen, wird eine Condition (*consumable*) verwendet. Der Consumer fragt zuerst ab, ob in allen Buffern eine Zahl vorhanden ist (*peek* – Methode). Wenn ein Buffer leer ist, ruft der Consumer die *await* – Methode auf und wartet damit, bis ein Producer eine weitere Zahl produziert hat und die *signalAll* – Methode aufruft. Erst danach fängt er wieder an, die Buffer auf vorhandene Zahlen zu testen.

Um ein Verhungern eines Consumers zu verhindern, wenn weitere Consumer dazukommen, haben wir nach jedem Konsum eine kurze Wartezeit eingebaut. Damit unterbricht der Consumer seinen Konsum kurz und die anderen Consumer haben eine Chance, CPU-Zeit zu bekommen und Zahlen zu konsumieren.

Wir haben die Buffer-Methoden mit *synchronized* gegen unerwünschte Nebeneffekte geschützt.

Das Warten auf die Befüllung eines leeren Buffers sowie die Synchronisation zwischen einzelnen Consumern haben wir mit einem Lock (*lock*) und einer Condition (*consumable*) gelöst.

Eine weitere Möglichkeit wäre noch, die Synchronisation mit *wait* und *notify/notifyAll* zu lösen.

Aufgabe 3

a)

In der Methode main() der Klasse LazyInitRace werden 3 Threads erzeugt, die jeweils die getInstance() Funktion der Klasse LazyInitRace aufrufen. Die getInstance() Funktion erzeugt ein neues ExpensiveObject Objekt falls die Variable instance null referenziert. Andernfalls returniert die Funktion getInstance() die in der Variable instance gespeicherte Referenz auf das ExpensiveObject Objekt. Da aber das Erzeugen eines ExpensiveObject Objekts rechenintensiv ist, dauert es zu lange bis die Variable instance überhaupt erst beschrieben wird und alle 3 Threads bekommen somit jeweils eine neue Instanz des ExpensiveObject Objekts retourniert.

b)

Möglichkeit 1

Das Schlüsselwort synchronized bei der Funktion getInstance() der Klasse ThreadSafeInit1 stellt sicher, dass immer nur ein Thread darauf Zugriff hat. Somit kann es nicht passieren, dass die 3 Threads unterschiedliche Instanzen des Objekts ExpensiveObject retourniert bekommen.

Möglichkeit 2

Diese Möglichkeit nutzt Java's Static Class Loading Mechanismus aus um sicherzustellen, dass nur eine Instanz des ExpensiveObject Objekts erstellt wird. Da statische Klassen in Java erst geladen werden sobald z.B. auf eine Methode oder Variable dieser zugegriffen wird ist es möglich diese Funktionalität für Synchronisationszwecke auszunutzen. Beim Aufruf von getInstance() der Klasse ThreadSafeInit2 wird auf die Variable INSTANCE der statischen inneren Klasse Helper zugegriffen. Durch diesen Zugriff wird diese Variable einmalig instanziert. Das Schlüsselwort final ist nicht unbedingt notwendig, doch es stellt sicher dass es zu einem späteren Zeitpunkt nicht möglich ist, die Variable INSTANCE, eine andere Instanz vom Typ ExpensiveObject referenzieren zu lassen.