Übung N3: Weiterführende Konzepte

Patrick Bucher

17.05.2017

Inhaltsverzeichnis

1 Bounded Buffer **1**
Reflektion 1

2 Parkhaus **2**
Reflektion 2

3 Speed Count **3**
Reflektion 3

4 Suche nach grossen Primzahlen **3**
Reflektion 3

5 Optional: Container threadsicher machen **4**
b) Was stellen Sie fest? 4
c) 4
d) 4
e) 4

1 Bounded Buffer

Ich verwende eine `ArrayBlockingQueue`.

Reflektion

Frage: Worin liegt der grösste Unterschied, zwischen Ihrem Bounded Buffer und der gewählten Blocking Queue? Haben Sie Unterschiede in den Tests der beiden Aufgaben festgestellt?

Antwort: Mein `BoundedBuffer` verfügt über eine blockierende `get()`-Methode. Solange diese Methode keinen Wert aus dem Buffer entfernt und zurückgegeben hat, kann sie nicht

beendet werden. Die `ArrayBlockingQueue` hat als Gegenstück eine `poll()`-Methode, die sofort `null` zurückgibt, wenn keine Werte in der Datenstruktur gespeichert sind. Darum muss der Consumer überprüfen, wie viele Werte er tatsächlich schon ausgelesen hat.

```
while (itemsToConsume > 0) {
    String str = queue.poll();
    if (str != null) {
        Assert.assertNotNull(str);
        itemsToConsume--;
    }
}
```

Frage: Wie sind Sie mit den anders lautenden Methodennamen umgegangen?

Antwort: Da ich den Test für den Einsatz eines Executors ohnehin umschreiben musste, war das kein grosses Problem; ich ersetzte einfach die Methodennamen. Besser wäre es gewesen, wenn mein `BoundedBuffer` das Interface `Queue` implementiert hätte, dann wäre dieses Problem gar nicht aufgetreten.

2 Parkhaus

Reflektion

Frage: Wie viele Autos konnten Sie maximal nebenläufig "fahren" lassen?

Antwort: So viele, wie mein Computer Rechenkerne hat.

Frage: Wie effizient arbeitet Ihre Simulation?

Antwort: Das hängt von den Simulationsparametern ab.

Frage: Wie fair sind die Parkhäuser?

Antwort: Die Autos dürfen in der Reihenfolge ihrer Ankunft parkieren. Ich halte das für fair. Man könnte aber auch ihre geplante Parkdauer berücksichtigen, sodass Kurzzeitparker Langzeitparkern gegenüber bevorzugt würden.

Frage: Falls Sie die Stadtplanung unterstützt haben: Was raten Sie den Stadtplanern von "Pannobile", welche Kapazitäten sollen die Parkhäuser haben?

Antwort: Da es viele Fahrer gibt, die nur das erstbeste Parkhaus ansteuern wollen, aber in meiner Simulation alle Parkhäuser gleich gross sind, muss man sich mit einer Quote von ca. 90% zufrieden geben, wenn man nicht alle Parkhäuser überdimensionieren möchte. Die Simulation müsste verschiedengrosse Parkhäuser zulassen, um eine bessere Aussage machen zu können.

3 Speed Count

Reflektion

Frage: Was können Sie über die Performance der beiden Thread-sicheren Zähler aussagen?

Antwort: Der `AtomicCounter` ist schneller.

Frage: Was stellen Sie bei den Testresultaten fest?

Antwort: Die Performanceunterschiede bewegen sich in einem Raum von 2 bis 15 Prozent.

Frage: Wie erklären Sie sich diese Testresultate?

Antwort: `synchronized` verlangsamt die Anwendung massiv.

Frage: Welche GEnauigkeit erreicht Ihr Test?

Antwort: Keine besonders hohe Genauigkeit, die Abweichungen bewegen sich in einem Raum von 2 bis 15 Prozent.

4 Suche nach grossen Primzahlen

Reflektion

Frage: Wie lange dauert es jetzt?

Antwort: (Ich vergleiche die beiden loggenden Implementierungen, nicht die synchrone loggende mit der asynchronen in-eine-Datei-schreibende.) Synchron: ca. zwei Minuten. Asynchron: ca. eine Minute.

Frage: Wie viele Threads lassen Sie laufen?

Antwort: Einen mehr als die Anzahl der Prozessor-Kerne meines Rechners.

Frage: Was passiert, wenn Sie die Anzahl Threads verdoppeln, vervierfachen, verzehnfachen?

Antwort: Es dauert tendenziell länger, wobei ich zu wenige Test laufen gelassen habe, um Zufall auszuschliessen. Mehr Threads benötigen jedoch mehr Verwaltungsaufwand, wodurch es tatsächlich länger dauern dürfte.

Frage: Können Sie die Applikation noch schneller machen?

Antwort: Auf meinem Rechner wohl nicht signifikant, ohne dabei Genauigkeit (bzw. hohe Wahrscheinlichkeiten für Primzahlen) zu opfern.

5 Optional: Container threadsicher machen

b) Was stellen Sie fest?

Hinweis: Als gemeinsame Datenstruktur für die Zahlen habe ich eine `ArrayList` verwendet.

Wird der Consumer vor den Producern gestartet, trifft er auf eine leere Liste (`iterator.hasNext() == false`) ist sieht seine Arbeit als beendet an.

Die Producers verursachen beim konkurrierenden Schreiben auf die `ArrayList` immer wieder eine `ArrayIndexOutOfBoundsException`. Das Problem ist wohl, dass ein Producer in die zu kleine Liste schreiben will, wenn sie noch nicht mit der Vergrösserung fertig ist.

Gelegentlich läuft der Test auch korrekt durch. Mit einer `LinkedList` erhalte ich hingegen eine `NullPointerException`.

c)

Mit der `CopyOnWriteArrayList` (die einzig threadsichere `List`-Implementierung) funktioniert es leider nicht. Das Problem ist wohl wieder, dass `hasNext()` “zu früh” ‘false" zurückliefert.

d)

Man müsste wohl eine nicht-threadsichere Queue einsetzen.

e)

Das ganze Design funktioniert nicht zuverlässig mit einer `BlockingQueue`. List man aus einer `BlockingQueue`, muss man wissen, wie viele Elemente man daraus herauslesen will, oder aber man unterbricht den Thread von aussen.

Besser wäre es, wenn man die `BlockingQueue` “schliessen” könnte, sobald keine weiteren Werte mehr hineingeschrieben werden müssen, wie man das mit Go Channels machen könnte.