8. Programmieraufgabe

Objektorientierte Programmiertechniken

LVA-Nr. 185.A01
2017/2018 W
TU Wien

Themen:

Exceptions, nebenläufige Programmierung

Ausgabe:

13. 12. 2017

Abgabe (Deadline):

20. 12. 2017, 12:00 Uhr

Abgabeverzeichnis:

`Aufgabe8`

Programmaufruf:

`java Test`

Grundlage:

Skriptum, Schwerpunkt auf den Abschnitten 4.1 und 4.2

Kontext

In der Vorweihnachtszeit blinken überall LED-Lichterketten. Diese sind zwar energiesparend, aber bei weitem nicht so romantisch wie echte Kerzen, die daher sehr gefragt sind. Deshalb herrscht in der Kerzenmanufaktur, wo Kerzen noch von Kerzenziehern in Handarbeit hergestellt werden, Hochbetrieb. Die Kerzenzieher stellen laufend neue Kerzen in zwei verschiedenen Größen her (kurz und lang) und deponieren sie im Kerzenlager. Die Verkäufer nehmen laufend Kerzen aus dem Lager und verkaufen sie. Im rechteckigen Lager benötigt eine kurze Kerze einen quadratischen Lagerplatz, eine lange Kerze zwei benachbarte quadratische Lagerplätze. Nachdem die Kerzenzieher eine neue Kerze produziert haben (das dauert einige Zeit), gehen sie durch das Lager, bis sie einen freien Platz finden, und deponieren dann dort die neu erzeugte Kerze. Wenn sie keinen freien Platz finden, warten sie eine kurze Zeit und gehen dann noch einmal durch das Lager (alternativ können sie auch von einem Verkäufer informiert werden, dass Kerzen verkauft wurden). Das wiederholen sie so oft, bis sie einen freien Platz gefunden haben. Die Verkäufer warten einige Zeit auf einen Kunden. Wenn dann Kunden ihren Kerzenwunsch (kurze oder lange Kerze) bekannt gegeben haben, gehen die Verkäufer durch das Lager, bis sie die passende Kerze gefunden haben. Falls die Verkäufer keine passende Kerze gefunden haben, warten sie eine kurze Zeit und gehen dann noch einmal durch das Lager (alternativ können sie auch von einem Kerzenzieher informiert werden, dass Kerzen produziert wurden). Die Produktion und der Verkauf gehen so lange, bis am Abend eine Glocke das Ende anzeigt. Dann stellen alle Kerzenzieher und Verkäufer ihre Aktivitäten ein.

Welche Aufgabe zu lösen ist

Simulieren Sie die Kerzenmanufaktur mittels eines nebenläufigen Java-Programms. Stellen Sie dabei jeden Kerzenzieher und jeden Verkäufer durch einen eigenen Thread dar. Jeder Kerzenzieher fertigt eine Kerze oder er wartet auf einen freien Lagerplatz. Jeder Verkäufer verkauft eine Kerze oder wartet auf eine Kerze. Stellen Sie das rechteckige Lager durch schachbrettartig angeordnete quadratische Lagerplätze dar. Eine kurze Kerze benötigt einen Lagerplatz, eine lange Kerze benötigt zwei nebeneinanderliegende Lagerplätze (Nord-Süd oder West-Ost). Stellen Sie sicher, dass mehrere Kerzenzieher gleichzeitig kurze wie lange Kerzen auf unterschiedlichen Lagerplätzen des Lagers deponieren können.

Damit die Simulation sehr viel schneller als in Wirklichkeit abläuft, lassen Sie die Kerzenzieher für die Produktion einer Kerze und einen Verkäufer für das Warten auf einen Kunden und das Verkaufsgespräch zufallsgesteuert wenige Millisekunden (5-50) warten. Simulieren Sie Wartezeiten mittels der Methode `Thread.sleep(n)`. Achtung: `sleep` behält alle Monitore (= Locks); Sie sollten `sleep` daher nicht innerhalb einer synchronized-Methode oder -Anweisung aufrufen, wenn während der

Wartezeit von anderen Threads aus auf dasselbe Objekt zugegriffen werden soll. Implementieren Sie nur eine der Alternativen für ein volles oder leeres Lager, entweder immer wieder warten und durchsuchen oder die Anderen informieren.

Wenn am Abend die Glocke läutet (1000 Millisekunden vergangen sind), geben Sie von allen Kerzenziehern und Verkäufern ihren Zustand (Kerze fertigen oder warten, Kerze verkaufen oder warten) und das Lager (wie weiter unten beschrieben) auf dem Bildschirm aus und beenden alle Threads. Verwenden Sie Thread.interrupt() um einen Thread zu unterbrechen.

Geben Sie immer, wenn ein Verkäufer eine Kerze verkauft hat, das gesamte Lager zeilenweise am Bildschirm aus. Verwenden Sie den Buchstaben "`+`" für einen Lagerplatz, auf dem eine kurze Kerze ist, den Buchstaben "`#`" für einen Lagerplatz, auf dem die Hälfte einer langen Kerze ist und den Buchstaben "`o`" für die freien Lagerplätze, zum Beispiel so:

```
# # + o # # +
+ o # + o o +
# # # + o + o
```

Die Klasse `Test` soll (nicht interaktiv) Testläufe der Simulation der Kerzenmanufaktur durchführen und die Ergebnisse in allgemein verständlicher Form in der Standardausgabe darstellen. Bitte achten Sie darauf, dass die Testläufe nach kurzer Zeit terminieren (maximal 10 Sekunden für alle zusammen). Bitte achten Sie darauf, dass die Testläufe keine Systemlimits wie maximale Anzahl an gleichzeitig aktiven Threads auf dem Abgaberechner (g0) überschreiten. Führen Sie mindestens drei Testläufe mit unterschiedlichen Einstellungen durch:

Für die Dauer des Kerzen-Produzierens und des Verkaufens sollen unterschiedliche Werte verwendet werden. Stellen Sie die Parameter so ein, dass irgendwann Kerzenzieher und irgendwann Verkäufer warten müssen (verändern sie dazu die Werte auch während eines Testlaufs).

Daneben soll die Klasse `Test.java` als Kommentar eine kurze, aber verständliche Beschreibung der Aufteilung der Arbeiten auf die einzelnen Gruppenmitglieder enthalten – wer hat was gemacht.

Aufgabenaufteilung beschreiben

Was im Hinblick auf die Beurteilung wichtig ist

Die insgesamt 100 für diese Aufgabe erreichbaren Punkte sind folgendermaßen auf die zu erreichenden Ziele aufgeteilt:

- Synchronisation richtig verwendet, auf Vermeidung von Deadlocks geachted, sinnvolle Synchronisationsobjekte gewählt, kleine Synchronisationsbereiche — 45 Punkte
- Lösung wie vorgeschrieben und sinnvoll getestet — 20 Punkte
- Zusicherungen richtig und sinnvoll eingesetzt — 15 Punkte
- Geforderte Funktionalität vorhanden (so wie in Aufgabenstellung beschrieben) — 15 Punkte

- Sichtbarkeit auf so kleine Bereiche wie möglich beschränkt 5 Punkte

Der Schwerpunkt bei der Beurteilung liegt auf korrekter nebenläufiger Programmierung und der richtigen Verwendung von Synchronisation sowie dem damit in Zusammenhang stehenden korrekten Umgang mit Exceptions. Punkteabzüge gibt es für

- fehlende oder fehlerhafte Synchronisation,
- zu große Synchronisationsbereiche, durch die sich Threads gegenseitig unnötig behindern (z.B. darf nicht das ganze Lager (alle Lagerplätze gleichzeitig) als ein einzelnes Synchronisationsobjekt blockiert werden, ausgenommen davon ist nur die Ausgabe der Lagerplätze am Bildschirm),
- nicht richtig abgefangene Exceptions im Zusammenhang mit nebenläufiger Programmierung,
- Nichttermination von `java Test` innerhalb von 10 Sekunden,
- unnötigen Code und mehrfache Vorkommen gleicher oder ähnlicher Code-Stücke,
- vermeidbare Warnungen des Compilers, die mit Generizität in Zusammenhang stehen,
- Verletzungen des Ersetzbarkeitsprinzips bei Verwendung von Vererbungsbeziehungen, mangelhafte Zusicherungen,
- schlecht gewählte Sichtbarkeit,
- unzureichendes Testen,
- und mangelhafte Funktionalität des Programms.

Wie die Aufgabe zu lösen ist

Überlegen Sie sich genau, wie und wo Sie Synchronisation verwenden. Halten Sie die Granularität der Synchronisation möglichst klein, um unnötige Beeinflussungen anderer Threads zu reduzieren (Es sollen gleichzeitig mehrere Kerzenzieher Kerzen einlagern und Verkäufer aus dem Lager entnehmen können). Vermeiden Sie aktives Warten, indem Sie immer `sleep` aufrufen, wenn Sie eine bestimmte Zeit warten müssen. Beachten Sie, dass ein Aufruf von `sleep` innerhalb einer `synchronized`-Methode oder -Anweisung den entsprechenden Lock nicht freigibt.

Testen Sie Ihre Lösung bitte rechtzeitig auf der g0, da es im Zusammenhang mit Nebenläufigkeit große Unterschiede zwischen den einzelnen Plattformen geben kann. Ein Programm, das auf einem Rechner problemlos funktioniert, kann auf einem anderen Rechner (durch winzige Unterschiede im zeitlichen Ablauf) plötzlich nicht mehr funktionieren. Stellen Sie sicher, dass die maximale Anzahl an Threads und der maximale Speicher nicht überschritten werden. Dazu ist es sinnvoll, dass Sie im Threadkonstruktor explizit die Stackgröße mit einem kleinen Wert angeben (z.B. 16k).

Nebenläufigkeit kann die Komplexität eines Programms gewaltig erhöhen. Achten Sie daher besonders darauf, dass Sie den Programm-Code so klein und einfach wie möglich halten. Jede unnötige Anweisung kann durch zusätzliche Synchronisation (oder auch fehlende Synchronisation) eine versteckte Fehlerquelle darstellen und den Aufwand für die Fehlersuche um vieles stärker beeinflussen als in einem sequentiellen Programm.

Warum die Aufgabe diese Form hat

Die Simulation soll die nötige Synchronisation bildlich veranschaulichen und ein Gefühl für eventuell auftretende Sonderfälle geben. Einen speziellen Sonderfall stellt das Simulationsende dar, das (aus Sicht eines Kerzenziehers oder eines Verkäufers) jederzeit in jedem beliebigen Zustand auftreten kann. Dabei wird auch geübt, nach einer an einer beliebigen Programmstelle aufgetretenen Exception den Objektzustand so weit wie nötig zu rekonstruieren, um ein sinnvolles Ergebnis zurückliefern zu können.

Was im Hinblick auf die Abgabe zu beachten ist

Gerade für diese Aufgabe ist es besonders wichtig, dass Sie (abgesehen von geschachtelten Klassen) nicht mehr als eine Klasse in jede Datei geben und auf aussagekräftige Namen achten. Sonst ist es schwierig, sich einen Überblick über Ihre Klassen und Interfaces zu verschaffen. Verwenden Sie keine Umlaute in Dateinamen. Achten Sie darauf, dass Sie keine Java-Dateien abgeben, die nicht zu Ihrer Lösung gehören (alte Versionen, Reste aus früheren Versuchen, etc.).

keine Umlaute