8. Programmieraufgabe

Objektorientierte Programmiertechniken

LVA-Nr. 185.A01
2020/2021 W
TU Wien

Themen:

Exceptions, nebenläufige Programmierung

Ausgabe:

02. 12. 2020

Abgabe (Deadline):

9. 12. 2020, 12:00 Uhr

Abgabeverzeichnis:

`Aufgabe8`

Programmaufruf:

`java Test`

Grundlage:

Skriptum, Schwerpunkt auf den Abschnitten 4.1 und 4.2

Kontext

Eine blühende Wiese benötigt einen humusreichen Boden. Viele Lebewesen tragen dazu bei, dass das organische Material in Humus umgewandelt wird und die Bodenschichten durchmischt werden. Regenwürmer fressen organisches Material und scheiden mit Mikroorganismen angereichertes verdautes Material wieder aus. Maulwürfe graben ein stark verzweigtes Röhrennetz in die Erde und durchmischen so, wie auch die Regenwürmer, die Bodenschichten. Regenwürmer sind auch die Lieblingsspeise von Maulwürfen. Maulwürfe graben den Regenwürmern nach und verscheuchen sie manchmal durch ihre Grabgeräusche an die Oberfläche.

Welche Aufgabe zu lösen ist

Simulieren Sie die Bewegungen eines Maulwurfs und die von mehreren Regenwürmern mittels eines nebenläufigen Java-Programms. Stellen Sie dabei den Maulwurf und mehrere Regenwürmer durch jeweils einen eigenen Thread dar. Um die Simulation stark zu vereinfachen wird das verzweigte Röhrennetz des Maulwurfs in ein rechtwinkeliges Röhrennetz in einer Ebene abgebildet. Das Röhrennetz hat keine Sackgassen. Der Maulwurf und die Regenwürmer bewegen sich ausschließlich in diesem Röhrennetz. Die Röhren können sich kreuzen. Daher gibt es Kreuzungen mit drei und mit vier abgehenden Röhren. Das Röhrennetz ist in viele gleich lange Röhrenabschnitte unterteilt.

Ein Maulwurf und ein Regenwurm belegen immer jeweils 2 Röhrenabschnitte (der Kopf belegt den ersten Röhrenabschnitt, der Körper oder das Ende den zweiten Röhrenabschnitt). Ein Maulwurf bewegt sich immer um einen Röhrenabschnitt in der Richtung des Kopfes weiter. Falls sich auf dem neuen Röhrenabschnitt des Kopfes ein Kopf oder Ende eines Regenwurms befindet, wird der Regenwurm gefressen. Dann wird der Thread des Regenwurms beendet und der Kopf und das Ende des Regenwurms aus dem Röhrennetz entfernt. Nachdem sich ein Maulwurf weiterbewegt hat, wartet er eine kurze Zeit, bevor er sich wieder weiterbewegt. Im Gegensatz zum Maulwurf kann sich ein Regenwurm jedes Mal nur um 2 Röhrenabschnitte auf einmal in Richtung des Kopfes weiterbewegen. Weiterbewegen kann sich der Regenwurm aber nur, falls die zwei Zielröhrenabschnitte frei sind (weder ein Regenwurm noch ein Maulwurf darf sich dort befinden). Nachdem sich ein Regenwurm zu den nächsten beiden Röhrenabschnitte weiterbewegt hat, wartet er eine kurze Zeit, bevor er sich wieder weiterbewegt. Bestimmen Sie durch (eine oder mehrere) einfache lokale Strategien, wohin sich ein Maulwurf oder ein Regenwurm an Kreuzungen weiterbewegt (Zufall, fixe Richtung, Maulwurf/Regenwurm auf dem Abschnitt, ...). Wenn keine zwei freien Röhrenabschnitte vorhanden sind, wartet der Regenwurm eine kurze Zeit. Alternativ kann sich der Regenwurm auch umdrehen (Kopf und Ende werden vertauscht), wenn er schon einmal gewartet hat. Zählen sie bei jedem Regenwurm mit, wie oft er gewartet hat und wie oft er sich von zwei Röhrenabschnitten zu den

Regenwurm muss sich um 2 Röhrenabschnitte weiterbewegen

nächsten zwei Röhrenabschnitten weiterbewegt hat. Alle Regenwürmer können sich gleichzeitig unabhängig vom Maulwurf und von den anderen Regenwürmern weiterbewegen.

Damit die Simulation viel schneller als in Wirklichkeit abläuft, lassen Sie Maulwurf und Regenwürmer zufallsgesteuert wenige Millisekunden (5-50) warten. Simulieren Sie Wartezeiten mittels `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.

Wenn der Maulwurf die maximale Anzahl an Warteschritten (32) erreicht hat, geben Sie vom Maulwurf und von allen Regenwürmern die Anzahl der Warteschritte / Kehrtwendungen, die Anzahl der Bewegungsschritte und eine Identifikation der beiden Röhrenabschnitte, auf denen sie sich befinden, auf dem Bildschirm aus und beenden Sie alle Threads. Verwenden Sie `Thread.interrupt()` um einen Thread abzubrechen.

Geben sie immer, nachdem der Maulwurf gewartet hat, das Röhrennetz zeilenweise am Bildschirm aus. Verwenden Sie den Buchstaben "`#`" für den Kopf eines Maulwurfs, den Buchstaben "`*`" für den Körper des Maulwurfs, den Buchstaben "`+`" für den Kopf eines Regenwurms, den Buchstaben "`-`" für ein waagrechtes Ende eines Regenwurms, den Buchstaben "`|`" für ein senkrechtes Ende eines Regenwurms und den Buchstaben "`0`" für einen Röhrenabschnitt, zum Beispiel so:

```
000000000000#*0000+
0   0   0          |
0   0   0          0
+-0000000000000000+-
0   0   0   0      0
0   |   00000-+0000
0   +   0          0
00000+-000000000000
```

Die Klasse `Test` soll (nicht interaktiv) Testläufe der Simulation vom Maulwurf und von den Regenwürmern 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). Und achten Sie auch 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 (drei unterschiedliche Röhrennetze, unterschiedliche Startpositionen für Maulwurf und Regenwürmer):

Für die Dauer des Weiterbewegens sollen für jeden Regenwurm unterschiedliche Werte verwendet werden. Stellen Sie die Parameter so ein, dass irgendwann Regenwürmer 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 (Regenwürmer) gegenseitig unnötig behindern (z. B. darf nicht das gesamte Röhrennetz – alle Röhrenabschnitte gleichzeitig – als ein einzelnes Synchronisationsobjekt blockiert werden, ausgenommen davon ist nur die Ausgabe des Röhrennetzes am Bildschirm),
- nicht richtig abgefangene Exceptions im Zusammenhang mit nebenläufiger Programmierung,
- Nichttermination von `java Test` innerhalb von 10 Sekunden,
- unnötigen Code und das 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.

keine zu große Synchronisationsbereiche

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 sich gleichzeitig mehrere Regenwürmer weiterbewegen können). Vermeiden Sie aktives Warten, indem Sie immer `sleep` aufrufen, wenn eine bestimmte Zeit gewartet werden muss. 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 Regenwurms) 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 (Röhre ist kein zulässiger Dateiname). Achten Sie darauf, dass Sie keine Java-Dateien abgeben, die nicht zu Ihrer Lösung gehören (alte Versionen, Reste aus früheren Versuchen, etc.).

keine Umlaute