9. Programmieraufgabe

Programmierparadigmen

LVA-Nr. 194.023
2023/2024 W
TU Wien

Themen:

nebenläufige Programmierung, Prozesse und Interprozesskommunikation

Ausgabe:

11. 12. 2023

Abgabe (Deadline):

8. 1. 2024, 14:00 Uhr

Abgabeverzeichnis:

`Aufgabe9`

Programmaufruf:

`java Test`

Grundlage:

Skriptum, Schwerpunkt auf den Abschnitten 5.3 und 5.4

Kontext

Die Myrmekologen wollen das Verhalten von Ameisenstaaten, insbesondere das globale Verhalten durch lokale Aktionen, nicht nur in einem Formicarium beobachten, sondern, da es einfacher ist, auch mit einem Computer simulieren.

Welche Aufgabe zu lösen ist

Simulieren Sie die Bewegungen der Ameisen eines Ameisenstaates von Blattschneiderameisen, die Blätter sammeln, mittels zwei Java-Prozessen, die von dem Testprogramm gestartet werden.

Der erste Prozess (Programm mit dem Namen `Arena`) simuliert die Bewegungen der Ameisen in der Arena. Stellen Sie dabei jede Ameise durch einen eigenen Thread dar. Zur Vereinfachung der Simulation wird die Arena nur in einer Ebene simuliert, d.h. 2-dimensional und nicht 3-dimensional. Die Ebene wird schachbrettartig in Felder geteilt. Die Anzahl der Ameisen in der Arena und die Länge und Breite der Arena (Anzahl der Felder) wird über Kommandozeilenargumente übergeben. Es wird eine entsprechend große Arena erzeugt und eine entsprechende Anzahl an Ameisen auf zufällige Positionen in der Arena platziert. Die Ameisen können sich nur in dieser Ebene bewegen und diese nicht verlassen. Eine Ameise muss sich immer gleichzeitig auf zwei benachbarten Feldern (zwei senkrecht oder waagrecht benachbarten Feldern) befinden, der Kopf auf einem Feld, der Körper auf dem anderen Feld. Die Ameise kann in vier verschiedene Richtungen ausgerichtet sein, nach links, rechts, oben oder unten. Es darf sich immer maximal eine Ameise (ein Ameisenteil) auf einem Feld befinden. Eine Ameise kann sich vorwärts bewegen, sich schräg links vorwärts bewegen, sich schräg rechts vorwärts bewegen, sich nach links bewegen oder nach rechts bewegen. Vorwärts bewegt sich die Ameise 2 Felder in Blickrichtung weiter. Bei einer Bewegung nach schräg links vorwärts wird der Körper auf das Feld links neben der alten Kopfposition bewegt, die neue Kopfposition ist dann in der unveränderten Blickrichtung das Feld vor dem Körper. Bei einer Bewegung nach schräg rechts vorwärts wird der Körper auf das Feld rechts neben der alten Kopfposition bewegt, die neue Kopfposition ist dann in der unveränderten Blickrichtung das Feld vor dem Körper. Beim nach links Bewegen dreht sie sich um 90 Grad nach links und landet auf den beiden Feldern, die sich im rechten Winkel zur alten Blickrichtung links neben dem Feld ihres Kopfes befinden. Beim nach rechts Bewegen dreht sie sich um 90 Grad nach rechts und landet auf den beiden Feldern, die sich im rechten Winkel zur alten Blickrichtung rechts neben dem Feld ihres Kopfes befinden. Beim nach links oder nach rechts Bewegen hat sich dann die Blickrichtung um 90 Grad geändert. Nachdem sich eine Ameise auf das nächste Feldpaar bewegt hat, wartet sie eine kurze Zeit, bevor sie sich weiterbewegt. Bestimmen Sie durch (eine oder mehrere oder gemischte) einfache lokale Strategien, wohin sich die Ameise weiterbewegt

(Zufall, Pheromonlevel, Feldpaar frei, ...). Wenn kein freies Feldpaar vorhanden ist, wartet die Ameise eine kurze Zeit. Zählen Sie bei jeder Ameise mit, wie oft sie gewartet hat und wie oft sie sich von einem Feldpaar zum nächsten bewegt hat. Alle Ameisen können sich gleichzeitig unabhängig von den anderen Ameisen bewegen.

In der Mitte der Arena befindet sich der Eingang zum Nest. Der Eingang belegt vier in einem Quadrat angeordnete Felder. Auf manchen Feldern befinden sich Blätter, die bei der Erstellung der Arena zufällig (passend zur Größe der Arena) platziert werden. Landet ein Körperteil einer Ameise auf einem Blatt, so schneidet die Ameise, sofern sie noch keinen Blattteil trägt, einen Teil von diesem Blatt ab und trägt ihn mit sich zum Eingang des Nests. Die Ameise weiß, wo sich das Nest befindet und wählt einen möglichst kurzen Weg zurück zum Nest. Auf dem Weg zurück zum Nest hinterlässt die Ameise eine Pheromonspur. Der Pheromonlevel, der für jedes einzelne Feld gespeichert wird, ist eine ganze Zahl zwischen 0 und 9. Auf dem Weg zurück zum Eingang wird der Pheromonlevel um 1 erhöht, sofern der alte Wert kleiner als 9 ist. Dieser Pheromonlevel kann für eine lokale Bewegungsheuristik der Ameise verwendet werden. Wenn eine Ameise den Eingang zum Nest erreicht hat, deponiert sie dort den Blattteil, den sie trägt, und macht sich dann wieder auf die Suche nach einem Blatt. Eine Ameise erreicht den Eingang, wenn alle 4 Felder des Eingangs nicht von einer fremden Ameise besetzt sind und die Ameise sich auf zumindest ein Feld des Eingangs bewegen kann.

Damit die Simulation sehr viel schneller als in Wirklichkeit abläuft, lassen Sie die Ameisen zufallsgesteuert wenige Millisekunden (5-50) warten (sowohl nach einer erfolgreichen Bewegung als auch beim Warten auf ein freies Feldpaar). 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 synchronized-Anweisung aufrufen, wenn während der Wartezeit von anderen Threads aus auf dasselbe Objekt zugegriffen werden soll.

Alle Testausgaben auf die Datei `test.out` schreiben

Wenn eine Ameise die maximalen Anzahl von 64 Warteschritten erreicht hat, geben Sie von allen Ameisen den Zählerstand der Warteschritte und die Anzahl der Bewegungungen und das Feldpaar (auf welchen Feldern sie sich befindet) in der Datei `test.out` aus und beenden alle Threads. Verwenden Sie Thread.interrupt() um einen Thread zu unterbrechen. Geben Sie die Positionen als X- und Y-Koordinaten der Felder sowie die Zählerstände aus, und beenden Sie den Thread. Die dargestellte Arenaebene darf in jeder Dimension (waagrecht und senkrecht) nicht mehr als 80 Felder haben.

Wählen Sie eine beliebige Ameise als Leitameise aus und schreiben Sie immer, nachdem die Leitameise gewartet hat, die Felder zeilenweise auf die Datei `test.out`. Verwenden Sie das Zeichen „`<`“ für einen Kopf, der nach links blickt, das Zeichen „`>`“ für einen Kopf, der nach rechts blickt, den Buchstaben „`A`“ für einen Kopf, der nach oben blickt, den Buchstaben „`V`“ für einen Kopf, der nach unten blickt, für den Rumpf (egal in welcher Richtung die Ameise blickt) das Zeichen „`+`“, ein Leerzeichen für ein freies Feld mit Pheromonlevel 0, eine der Ziffern „`1`“ bis „`9`“ für ein freies Feld mit Pheromonlevel 1 bis 9, den Buchstaben „`O`“ für den Eingang in das Nest und den Buchstaben „`X`“ für ein Blatt, zum Beispiel so:

```
<+777<+666<+
62  XXXXX
4         XX
<+3<+4002222A
5     00    +
5     2    33
+X    3  A33
VX44+>4+>+
```

Der zweite Prozess (Programm mit dem Namen `Nest`, der vom `Arena` Prozess gestartet wird) simuliert die Lagerung der Blattteile im Nest. Stellen Sie jeden Blattteil als ein eigenes Objekt dar. Jeder Blattteil hat eine Fläche, die als Gleitkommazahl gespeichert wird. Wenn eine Ameise einen Teil von einem Blatt abschneidet, soll ein Blattteilobjekt erstellt werden, dessen Fläche mit einem Zufallswert in einem kleinen, eingeschränkten Bereich gesetzt wird. `Nest` ist mit `Arena` durch eine Pipeline verbunden. Wenn eine Ameise den Eingang des Nests erreicht, deponiert sie den Blattteil dort. Das soll so simuliert werden, dass das Blattteilobjekt im `Arena`-Prozess serialisiert wird, an den `Nest`-Prozess weitergegeben, dort deserialisert und in einer Liste gespeichert wird. Wenn der `Arena`-Prozess terminiert, sollen die Blattteilobjekte für Menschen verständlich auf die Datei `test.out` geschrieben werden und dann auch der `Nest`-Prozess terminiert werden.

Die Klasse `Test` soll in einem eigenen Prozess (nicht interaktiv) Testläufe der Simulation von Ameisenstaaten durchführen und die Ergebnisse in allgemein verständlicher Form in die Datei `test.out` schreiben, indem von `Test` mehrere `Arena`-Prozesse mit unterschiedlichen Kommandozeilenargumenten erstellt werden. 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 Weiterbewegens sollen für jede Ameise unterschiedliche Werte verwendet werden. Stellen Sie die Parameter so ein, dass irgendwann Ameisen 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:

- Nebenläufigkeit und Synchronisation richtig verwendet, auf Vermeidung von Deadlocks geachted, sinnvolle Synchronisationsobjekte gewählt, kleine Synchronisationsbereiche — 30 Punkte
- Prozessverwaltung und Interprozesskommunikation richtig verwendet — 20 Punkte

- Lösung wie vorgeschrieben und sinnvoll getestet 20 Punkte
- Geforderte Funktionalität vorhanden (so wie in Aufgabenstellung beschrieben) 15 Punkte
- Zusicherungen richtig und sinnvoll eingesetzt 10 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, korrekter Prozessverwaltung und Interprozesskommunikation, 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 (Ameisen) gegenseitig unnötig behindern (z. B. darf nicht die gesamte Arena oder zwei Zeilen oder Spalten der Arena – alle Felder gleichzeitig – als ein einzelnes Synchronisationsobjekt blockiert werden, ausgenommen davon ist nur die Ausgabe der Arena in die Datei `test.out`),
- 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 Ameisen 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 einer Ameise) jederzeit in jedem beliebigen Zustand auftreten kann. Dabei wird auch geübt, wie nach einer an einer beliebigen Programmstelle aufgetretenen Exception der Objektzustand so weit wie nötig rekonstruiert wird, dass ein sinnvolles Ergebnis zurückgeliefert werden kann.

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