9. Programmieraufgabe

Programmierparadigmen

LVA-Nr. 194.023
2024/2025 W
TU Wien

Themen:

nebenläufige Programmierung, Prozesse und Interprozesskommunikation

Ausgabe:

16. 12. 2024

Abgabe (Deadline):

20. 1. 2025, 14:00 Uhr

Abgabeverzeichnis:

`Aufgabe9`

Programmaufruf:

`java Test (keine Argumente, nicht interaktiv)`

Grundlage:

Skriptum, Schwerpunkt auf den Abschnitten 5.3 und 5.4

Kontext

Bei Gebäuden ist es wichtig, dass sie auch im Fall einer Katastrophe wie zum Beispiel eines Brandes sicher sind. Eine ausreichende Kapazität der Fluchtwege ist dabei besonders wichtig. Um eine bessere Einschätzung der richtigen Dimensionierung der Fluchtwege zu erhalten, soll die Effektivität der Fluchtwege, die zu einem sicheren Sammelpunkt führen, in einer Simulation evaluiert werden. Um die Simulation im Rahmen dieser Lehrveranstaltung entwickeln zu können, wird die Realität stark vereinfacht.

Welche Aufgabe zu lösen ist

Simulieren Sie die Bewegungen von Personen, die sich auf einem Netz von Fluchtwegen innerhalb eines Gebäudes zu einem Sammelpunkt bewegen, mittels zwei Java-Prozessen, die von dem Testprogramm gestartet werden. Der erste Prozess (Programm mit dem Namen `Simulation`) simuliert die Bewegungen von Personen auf dem Wegenetz eines Gebäudes. Der zweite Prozess (Programm mit dem Namen `Sammelpunkt`) speichert Informationen über Personen, die den Sammelpunkt erreicht haben. Stellen Sie dabei jede Person durch einen eigenen Thread dar.

Zur Vereinfachung der Simulation wird das rechtwinkelige Wegenetz nur in einer Ebene simuliert (nicht mehrstöckig). Die Wege können nur parallel oder im rechten Winkel zueinander verlaufen. Es gibt keine Einschränkungen in Bezug auf Kreuzungen, Verzweigungen oder Vereinigungen von Wegen. Zur Simulation werden die Fluchtwege in quadratische Felder geteilt. Die Länge oder die Breite eines Weges muss immer ein Vielfaches der Seitenlänge eines Feldes betragen. Ein Weg muss mindestens zwei Felder breit sein. Es gibt genau einen Sammelpunkt. Der Zugang zum Sammelpunkt wird durch spezielle Zugangsfelder am Ende des Fluchtwegs gekennzeichnet. Es müssen mindestens zwei Zugangsfelder existieren und diese müssen aneinander grenzen. Es muss möglich sein, von jedem Feld des Wegenetzes zum Sammelpunkt zu gelangen.

Erstellen Sie mindestens drei unterschiedliche Fluchtwegenetze mit einem eindeutigem Namen. Die Anzahl der Personen und der Name eines Wegenetzes wird über Kommandozeilenargumente an den `Simulation`-Prozess übergeben. Es wird dann eine entsprechende Anzahl von Personen auf zufällige Positionen in dem Wegenetz positioniert. Eine Person muss sich immer gleichzeitig mit beiden Füßen auf zwei an den Seiten angrenzenden Feldern befinden, der linke Fuß auf einem Feld, der rechte Fuß auf dem anderen Feld. Die Person kann in vier verschiedene Richtungen ausgerichtet sein (siehe Beispiel weiter unten). Es darf sich immer maximal eine Person (ein Fuß) auf einem Feld befinden. Die Personen bewegen sich auf dem Wegenetz weiter. Bei so einem Bewegungsschritt müssen beide Füße auf einem beliebigen an den Seiten oder Ecken benachbartem Feld neu positioniert werden. Die beiden Felder, auf denen sich die Person befunden hat, sind als neue Position nicht zulässig. Es gibt keine

Einschränkungen bezüglich der Richtung der Person auf den neuen Feldern. Folgendes Bild beschreibt mit den Buchstaben „`L`“ (linker Fuß) und „`R`“ (rechter Fuß) alte Positionen, mit den Buchstaben „`l`“ (linker Fuß) und „`r`“ (rechter Fuß) neue Positionen, der Buchstabe „`x`“ beschreibt alle möglichen neuen Positionen. Die ersten vier Beispiele zeigen alle möglichen Ausgangspositionen mit allen gültigen Nachbarfeldern, die anderen fünf Beispiele zeigen einige mögliche Bewegungen:

```
xxxx  xxx   xxxx  xxx
xLRx  xLx   xRLx  xRx
xxxx  xRx   xxxx  xLx
      xxx         xxx

lrxx  xxxx  xxxx  lxx  lrx
xLRx  xLRx  xLRx  rLx  xRx
xxxx  xlrx  xrlx  xRx  xLx
                  xxx  xxx
```

Beide Füße müssen gleichzeitig neu positioniert werden. Der Positionswechsel darf nicht in zwei Schritte aufgeteilt werden.

Bestimmen Sie durch (eine oder mehrere oder gemischte) einfache lokale Strategien, wohin sich eine Person weiterbewegt (Weg Richtung Sammelpunkt, Zufall, Feldpaar frei, ...). Wenn kein freies Feldpaar vorhanden ist, wartet die Person eine kurze Zeit. Zählen Sie bei jeder Person mit, wie oft sie gewartet hat und wie oft sie sich von einem Feldpaar zum nächsten bewegt hat. Alle Personen können sich gleichzeitig unabhängig von den anderen Personen bewegen.

Damit die Simulation sehr viel schneller als in Wirklichkeit abläuft, lassen Sie die Personen 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 alle Personen den Sammelpunkt erreicht haben oder wenn eine Person die maximale Anzahl von 64 Warteschritten (= Aufrufe von `sleep`) erreicht hat, geben Sie von allen Personen den Zählerstand der Warteschritte und die Anzahl der Bewegungen 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 der Felder (z.B. die X- und Y-Koordinaten in einer Ebene) sowie die Zählerstände aus, und beenden Sie den Thread. Das dargestellte Wegenetz darf in jede Richtung (Nord/Süd, Ost/West) nicht mehr als 80 Felder haben.

Wählen Sie eine beliebige Person als Leitperson aus und schreiben Sie immer, nachdem die Leitperson gewartet (`sleep` aufgerufen) hat, die Felder zeilenweise auf die Datei `test.out`. Geben Sie für Felder, auf denen sich ein linker Fuß befindet, das Zeichen „`L`“ aus, für Felder, auf denen sich ein rechter Fuß befindet, das Zeichen „`R`“ und für Felder, die ein Zugangsfeld zum Sammelpunkt sind, das Zeichen „`S`“. Geben Sie für die

restlichen Felder die Richtung des kürzesten Weges zum Sammelpunkt an (bei mehreren möglichen Richtungen eine beliebige dieser Richtungen). Verwenden Sie das Zeichen „<“ für die Richtung nach Westen, das Zeichen „>“ für die Richtung nach Osten, das Zeichen „^“ für die Richtung nach Norden und das Zeichen „v“ für die Richtung nach Süden, z.B. so:

```
vvvv    vvvv    vvvv
vvvv    vvvv    vvvv
vvv>>L>>vRLv<<<<<vvv
vv>>>R>>vvvv<<<<<<vv
v>>>>>>>vvvv<<R<<<<v
>>>>>>>>vvvv<<L<<<<<
        SSSS
```

oder so

```
vv   vv   vv
RL   vv   vv
>>>>>>>>L>>>>>S
>>>>>>>>R>>>>>S
^^   ^^
LR   ^^
```

Der zweite Prozess (Programm mit dem Namen `Sammelpunkt`, der vom `Simulation` Prozess gestartet wird) simuliert, wie sich die Personen im Sammelpunkt versammeln und speichert die Daten der Personen ab, die den Sammelpunkt erreicht haben. Dazu werden folgende Personendaten gespeichert: eine eindeutige Identifikationsnummer, die Anzahl der ausgeführten Bewegungsschritte auf dem Fluchtweg und die Anzahl der ausgeführten Warteschritte. Stellen Sie die Personendaten als ein eigenes Objekt dar. `Sammelpunkt` ist mit `Simulation` durch eine Pipeline verbunden. Wenn eine Person den Zugang zum Sammelpunkt erreicht hat (mit mindestens einem Bein auf einem Sammelpunktfeld steht), werden die Personendaten gespeichert. Das soll so simuliert werden, dass das Personendatenobjekt im `Simulation`-Prozess erstellt und dann serialisiert wird, an den `Sammelpunkt`-Prozess weitergegeben, dort deserialisert und in einer Liste gespeichert wird. Nachdem das Personendatenobjekt weitergegeben wurde, wird der Personenthread beendet und die Person aus dem Wegenetz entfernt. Wenn der `Simulation`-Prozess terminiert, sollen die Personendatenobjekte für Menschen verständlich auf die Datei `test.out` geschrieben werden und dann auch der `Sammelpunkt`-Prozess terminiert werden.

Die Klasse `Test` soll in einem eigenen Prozess (nicht interaktiv) Testläufe der Fluchtwegesimulation durchführen und die Ergebnisse in allgemein verständlicher Form in die Datei `test.out` schreiben, indem von `Test` mehrere `Simulation`-Prozesse mit unterschiedlichen Kommandozeilenargumenten erstellt werden. Test selbst darf keine Kommandozeilenargumente benötigen. Die Datei `test.out` soll in den Ordner geschrieben werden, in dem Test aufgerufen wird. Treffen Sie keine Annahmen, wie der Ordner heißt oder wie der Pfad zu diesem Ordner ist, indem der Aufruf von Test erfolgt (Aufgabe9 ist auf jeden Fall nicht der korrekte

Keine Annahmen über den Namen des Testverzeichnisses treffen

Verzeichnisname). 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 Person unterschiedliche Werte verwendet werden. Stellen Sie die Parameter so ein, dass irgendwann Personen 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 — 25 Punkte
- Lösung wie vorgeschrieben und sinnvoll getestet — 20 Punkte
- Geforderte Funktionalität vorhanden (so wie in Aufgabenstellung beschrieben) — 15 Punkte
- Kommentare richtig und sinnvoll eingesetzt — 10 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, fehlerhafte oder überflüssige Synchronisation,
- zu große Synchronisationsbereiche, durch die sich Threads (Personen) gegenseitig unnötig behindern (z. B. darf nicht das gesamte Wegenetz oder ganze Zeilen oder Spalten des Wegenetzes – alle Felder gleichzeitig – als ein einzelnes Synchronisationsobjekt blockiert werden, ausgenommen davon ist nur die Ausgabe des Wegenetzes in die Datei `test.out`),

  keine zu große Synchronisationsbereiche
- nicht richtig abgefangene Exceptions im Zusammenhang mit nebenläufiger Programmierung,
- Nichttermination von `java Test` innerhalb von 10 Sekunden,
- vermeidbare Warnungen des Compilers, die mit Generizität in Zusammenhang stehen,

- 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 sich gleichzeitig mehrere Personen 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 Person) 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, so 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