Nebenläufige Programmierung
Praktisches Projekt

Dokumentation

15. August 2012

Inhaltsverzeichnis

1 Die Klasse Professor

Der **Professor** ist die Main-Klasse und lässt sich in drei Phasen unterteilen: die Initialisierungsphase, die Hauptphase und die Terminierungsphase.

1.1 Die Initialisierungsphase

In der Initialisierungsphase werden abhängig von der Anzahl der Klausuren und Aufgaben zunächst Klausuren erstellt und auf die Stapel verteilt. Dies geschieht in **divide()**. Danach erstellt der Professor mit Hilfe von **initialize()** Threads in Anzahl der Assistenten und startet diese.

1.2 Die Hauptphase

Der Kern des Programms. In dieser Phase wird der Professor immer wieder abwechselnd zwei Aufgaben erledigen:

1.2.1 Die Endkontrolle der Klausur

Die Endkontrolle erfolgt in zwei Schritten: zunächst nimmt der Professor eine Klausur von seinem finalstack (1). Auf dieser ruft er dann **finish()** auf (2). Der finalstack ist hierbei als LinkedBlockingDeque realisiert, denn diese Datenstruktur ist thread-safe, was Professor und Assistenten ermöglicht, nebenläufig auf den finalen Stack zuzugreifen.
Enthält der Stapel des Professors keine Klausuren mehr, so überprüft er die Stapel der Assistenten. Enthalten auch diese keine Klausuren, so kann das Programm eventuell terminieren (siehe 1.3 und 6.2).

1.2.2 Das Umverteilen der Klausuren

Nachdem der Professor eine Klausur des finalstacks abgearbeitet hat, ruft er redistribute() auf (3), mit der er Klausuren zwischen den Stapeln umverteilt. Die **redistribute()**-Methode wird, wenn sie aufgerufen wird, zunächst feststellen, ob sie überhaupt Arbeit verrichten soll oder nicht. Dazu überprüft sie zunächst einen Parameter, der die Häufigkeit festlegt, mit der sie arbeiten soll (im Verhältnis zu den Aufrufen von **finish()**). Danach prüft sie, ob gerade ein Assistent schläft. Falls alle Assistenten arbeiten, ist ein Umverteilen wenig sinnvoll, sofern der Professor selbst noch Arbeit hat, und wird deshalb nicht durchgeführt. Das Umverteilen selbst passiert dann wie folgt:
Zunächst holt sich der Professor die Größe aller **ExamStacks**. Diese kann sich danach zwar ändern, dient aber nicht als sicherheitsrelevante Information, sondern nur als Anhaltspunkt, um das Verteilen effizient zu gestalten.

Die Stapel werden dann entsprechend ihrer Größe sortiert und die Durchschnittsgröße aller Stapel wird ermittelt. Dann werden vom größten Stapel so lange Klausuren genommen, bis er entweder keine mehr hergibt oder seine (geschätzte) Größe unter den Durchschnitt fällt. Jedes mal wenn dabei eine Klausur von Stapel genommen wird, wird seine Größenabschätzung entsprechend angepasst.
Für jede Klausur werden dann die zu erledigenden Aufgaben ermittelt. Dann wird diese Klausur auf den kleinsten Stapel gepackt, an welchem ein Assistent sitzt, der eine Aufgabe korrigiert, die in der Klausur noch zu erledigen ist. Falls alle Aufgaben der Klausur bis auf die, die auf dem größten Stapel korrigiert wird, schon erledigt sind, kommt die Klausur in einen Puffer und wird am Ende der Prozedur wieder auf den größten Stapel gelegt.
Am Ende der Umverteilung wird auf allen Klausurstapeln, die neue Klausuren erhalten haben, ein **distribute()** erzwungen. Falls der Stapel vorher leer war wird so wieder etwas auf den internen AssiStack gelegt und der schlafende Assistent geweckt.

1.3 Die Terminierungsphase

Nach jedem Durchlauf überprüft der Professor, ob die Arbeit beendet sein könnte. Dazu liest er ein globales Counter-Objekt namens waitingAssistants aus. Dieses Counter-Objekt ist eine Instanz der Klasse **IdleAssistantsCounter**. Zur Funktionsweise dieser Klasse siehe 4). Wenn der Professor dem Counter entnimmt, dass alle Assistenten „arbeitslos“ sind, so beendet er seine Hauptphase.
Kann der Professor mitsamt seinen Assistenten nun terminieren? Nein, denn es kann der Fall eintreten, dass ein Assistent erst nach der Überprüfung **increment()** aufruft, also wieder Arbeit hat, der Professor dies aber nicht erfährt und zu früh terminiert. Der Counter garantiert also nicht die Sicherheit des Programms!
Der Professor muss also in zwei Fällen entscheiden, ob das Programm terminieren darf:

(1) Er hat festgestellt, dass alle Assistenten warten
(2) Der Klausurstapel des Professors und alle anderen Klausurstapel waren leer (siehe 1.2.1)

Beides sind starke Anzeichen dafür, dass alle Klausuren abgearbeitet wurden, aber sie garantieren nicht die Sicherheit des Programms! Der Professor muss also noch einmal eine Überprüfung vornehmen. Dies geschieht in der Methode **testForTermination()**. Hierin ruft der Professor auf jedem

Assistenten-Thread **interrupt()** auf (4). Ein Assistent hat eine Schleife, in welcher er seine Arbeit erledigt, die so lange ausgeführt wird, wie er nicht interrupted ist. Wird der Assistent nun interrupted, beendet er zunächst seine aktuelle Arbeit, verlässt dann die Schleife und ruft die Methode **countdownLatch()** im Professor auf, welche ein Latch herunterzählt. Danach begibt er sich ins Wait-Set des Professors.
Der Professor wartet, bis alle Threads das Latch heruntergezählt haben (5). Damit weiß er, dass keiner der Assistenten mehr eine Klausur hat. Er überprüft also, ob alle Klausurstapel leer sind. Wenn das der Fall ist, setzt er ein globales Boolean-Flag (6) und benachrichtigt alle Assistenten (7).
Die Assistenten überprüfen nach dem **wait()** in einer äußeren Schleife dieses Flag und terminieren, falls es gesetzt ist. Ansonsten arbeiten sie weiter wie zuvor. Der Professor führt seine restlichen Endkorrekturen durch (8) und, so die Arbeit erledigt ist, terminiert, nachdem alle seine Assistenten terminiert sind (9).
Es ist wichtig, dass der Professor als Letztes terminiert, da die Assistenten zur Überprüfung des Programmzustandes noch eine Professor-Instanz benötigen.

2 Die Klasse Assistant

Der **Assistant** ist recht einfach gehalten und wird einfach eine Klausur nach der anderen von seinem rechten Stapel nehmen und kontrollieren, wobei er eine vollständig bearbeitete Klausur auf den Stapel des Professors legt (1) und alle anderen auf den Stapel seines linken Nachbarn (2). Die eigentlichen Operationen dafür werden vom ExamStack zur Verfügung gestellt. Vor allem betritt der Assistant während der Arbeit selbst niemals ein Wait-Set.

3 Die Klasse Exam

Die zur Verfügung gestellte Klasse Exam wird erweitert durch ein Boolean-Array correctedExersise, das kennzeichnet, welche Aufgaben bereits kontrolliert wurden. Die Korrektur-Methode wird bei einer bereits korrigierten Aufgabe sofort returnen.
Zudem hat das Exam eine Methode **isCorrected()**, mit der die Assistenten prüfen können, ob eine Klausur bereits vollständig korrigiert wurde. Die Methode **exercisesToDo()** wird vom Professor beim Umverteilen verwendet und gibt eine Liste mit den Aufgaben zurück, die noch nicht korrigiert wurden.
Schließlich wurden alle Methoden, die von mehreren Threads verwendet werden, mit dem synchronized-keyword geschützt, mehr dazu in Abschnitt 6.1.

4 Die Klasse ExamStack

4.1 Das generelle Konzept

Der ExamStack dient uns als Datenstruktur, mit der Assistenten und Professor effizient und nebenläufig sicher auf Klausuren zugreifen können. Das Ziel ist hierbei, dass der Professor den Stapel durchsehen kann (um ggf. umzuverteilen), ohne die Assistenten zu behindern. Darum werden die Klausurenstapel intern in zwei Stapel **assiStack** und **profStack** aufgeteilt. Assistenten und Professor arbeiten mit unterschiedlichen Zugriffsmethoden. Diese Methoden locken jeweils nur den Stapel, auf dem sie operieren.
Allen Methoden ist gemein, dass sie jeweils Klausuren vom Stapel entfernen, bevor sie ausgegeben werden. Sofern die Stapel also korrekt gelockt werden, ist es nicht möglich, dass zwei Threads gleichzeitig auf der selben Klausur arbeiten. Für das Locken der Stapel werden explizite Locks verwendet, das Lock des Assistentenstapels hat zusätzlich noch eine Condition. Da die Methoden von ExamStack von entscheidener Wichtigkeit für die Sicherheit und Lebendigkeit unserer Implementierung sind, werden sie im Folgenden genauer vorgestellt.

4.2 Die Methoden des AssisStacks

public Exam assiPop(): Wird vom Assistenten verwendet, um eine Klausur vom Stapel zu nehmen. Die Methode wird zunächst den Assistentenstapel locken und überprüfen, ob der Stapel leer ist. Wenn der Stapel leer ist, ruft die Methode **distribute()** auf. Ist der Stapel danach immer noch leer,erhöht sie den Counter waitingAssistants im Professor (mittels der Methode **increment()**). Dann ruft die Methode auf der Condition des Locks **await()** auf, um zu warten, bis wieder jemand etwas auf den Stapel legt. Ist der **await()**-Befehl abgearbeitet, wird waitingAssistants wieder um eins erniedrigt.
Dann wird wieder überprüft, ob der Stapel leer ist (while-Schleife). Wenn der Stapel mindestens ein Element hat, wird das oberste Element erst entfernt (1) und dann zurückgegeben (2). Danach wird das Lock des Assistentenstapels wieder freigegeben.

public void assiPush(Exam exam): Wird vom Assistenten verwendet, um eine Klausur auf den Stapel zu legen. Zunächst wird der Assistenten-Stapel gelockt, dann die Klausur daraufgelegt, dann auf der dazugehörigen Condition **signal()** aufgerufen und dann das Lock zurückgegeben.

4.3 Die Methoden des ProfStacks

public Exam ProfPull(): Lockt den Prof-Stapel, entfernt das oberste Element, gibt es zurück und gibt das Lock wieder frei. Falls der Prof-Stapel leer ist, wird zunächst **distribute()** aufgerufen. Wie in **assiPop()** kann es auch hier passieren, dass nach **distribute()** immer noch keine Klausur vorhanden ist. Anders als in **assiPop()** warten wir hier aber nicht auf eine mittels einer Condition, sondern geben in diesem Fall einfach **null** zurück.

public void profPush(Exam exam): Arbeitet analog zu **assiPush()**, nur dass keine Condition verwendet wird.

4.4 Die interne Umverteilung der Klausuren

public void distribute(): Verteilt die Klausuren auf die internen Stapel. Zunächst nimmt die Methode sich beide Locks. Dann überprüft sie, welcher der Stapel leer ist. Sind beide leer, returnt die Methode einfach (3). Ist keiner leer, ebenfalls (4). Ist der assiStack leer, so werden mittels **fillAssiStack()** Klausuren darauf gepackt. Ist der profStack leer, so bekommt der profStack alle Klausuren des assiStacks bis auf BUFFER_SIZE Klausuren. BUFFER_SIZE ist hier eine Konstante der Datenstruktur, die bei uns den Wert 5 hat, jedoch beliebig anders gewählt werden könnte. Am Ende werden beide Locks zurückgegeben.
Generell ist der Zweck von **distribute()** also, immer möglichst viele Klausuren auf den profStack zu legen, damit der Professor sich genug Klausuren zur Umverteilung nehmen kann, ohne den Assistenten bei der Arbeit zu stören.

private void fillAssiStack(): Eine lokale Hilfsmethode, die nur dazu da ist, das Auffüllen des assiStacks auszulagern. Nimmt sich beide Locks (eigentlich nicht notwendig, aber macht das Programm gegenüber Erweiterungen sicherer), geht mit einem Iterator über den profStack und verteilt so viele Klausuren auf den assiStack um, wie von unserer Konstante BUFFER_SIZE angegeben wird. Am Ende gibt sie beide Locks zurück.

5 Die Klasse IdleAssistantsCounter

Der **IdleAssistantsCounter** ist nichts anderes ist als ein als Monitor realisierter Zähler, den die ExamStacks bei Betreten/Verlassen eines Waitsets inkrementieren (**increment()**) bzw. dekrementieren (**decrement()**) und der gegenseitigen Ausschluss realisiert. Es gibt an, wieviele der Assistenten zum Zeitpunkt der Abfrage keine Klausuren zur Bearbeitung haben. Der Professor überprüft permanent den Counter und gerät in die Terminierungsphase, wenn der Counter auf dem Maximalwert steht (Anzahl aller Assistenten).

6 Behobene Probleme seit dem letzten Review

6.1 Mögliche Speicherinkonsistenz im Exam

In der Klasse **Exam** waren Methoden nicht synchronisiert, die nebenläufig von verschiedenen Threads verwendet wurden, z.B. die **correct()** Methode. Wir unterließen die Synchronisierung aus der Überlegung heraus, dass unsere Datenstruktur ExamStack sicherstellt, dass immer nur ein Thread eine Referenz auf ein Examen hat und nie zwei gleichzeitig. In der reinen Theorie dürfte es also nicht zu Data-Races kommen. Allerdings haben wir in der 17ten Vorlesung gelernt, dass die Praxis ein wenig anders aussieht (The Java Horror Picture Show).
Durch die Speicherarchitektur von Mehrkernprozessoren kann es passieren, dass ein Examen bei mehrenen Prozessoren zugleich im Cache liegt und somit ein Thread auf einer veralteten Version operiert. Dieses lässt sich nur verhindern, indem man erzwingt, dass das Objekt im Prozessorcache mit dem Objekt im Hauptspeicher synchronisiert wird. Dazu benötigt man Fences, die den Kritischen Abschnitt, in unserem Fall die gesamte Methode, einkapseln. Dadurch baut man sogenannte "Happens Before"-Beziehungen auf, die sicherstellen, dass ein Abschnitt vor einem anderen passiert.
Eine einfachere und sicherere Methode dazu stellt in Java das Schlüsselwort **synchronized** dar, welches genau diesen Effekt hat:
"There are several actions that create happens-before relationships. One of them is synchronization, as we will see in the following sections." [1]
"Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods. " [2]
Um also das Problem der möglichen Speicherinkonsistenz, die in letzter Konsequenz auch zu Data Races führen kann, zu beheben, genügt es, sämtliche Methoden, die von verschiedenen Threads aufgerufen werden, mittels **synchronized** zu schützen.
Einige Durchläufe (ca. 10000) mit verschiedenen Anzahlen von Examen (zwischen 100 und 10000) und Aufgaben (zwischen 5 und 42) haben gezeigt, dass gegenüber der alten Version des Programms sich keine nennenswerte Performanceänderung ergab. Zwei Beispieltestlogs finden sich in dem Ordner testlogs, es wurden jedoch erheblich mehr Test durchgeführt, deren logs nicht alle

[1] Zitiert nach: http://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html
[2] Zitiert nach: http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

gespeichert wurden. Die Klasse Testing, die sich bei den Sourcefiles befindet zeigt auf, wie die Tests durchgeführt wurden.
Auch Profiling und ein Blick auf die Threads ergab keine zusätzliche Sequenzialisierung. Dies war zu erwarten, da Examen nicht besonders häufig neu aus dem Speicher geladen werden müssen und der Hauptteil der Zeit in der correct- Methode verbraucht wird. Einige zusätzliche Speicherzugriffe, die durch das **synchronized** eventuell erzwungen werden, fallen also nicht ins Gewicht.

6.2 Das Busy-wait im Professor

Bisher hatten wir im Professor ein verstecktes Busy-Wait. Um vom finalstack Exams zu entfernen, verwendeten wir die Methode **pollFirst()** der LinkedBlockingDeque. Dies ist problematisch, da diese Methode nicht blockiert. War der Stapel also leer, so ging der Professor nicht etwa ins Wait-Set, sondern arbeitete weiter und fragte immer wieder ab, ob der finalstack eine Klausur enthält. Dadurch verbrachte der Professor viel Zeit damit, immer wieder die Schleife zu durchlaufen, obwohl noch kein neues Examen auf dem finalstack lag, weil die Assistenten alle noch am zeitaufwendigen Korrigieren waren. Diese Zeit war also verschwendet, da sich der Professor besser ins Wait-set begeben hätte sollen, um so für andere Threads Prozessorzeit freizugeben.
Dies haben wir jetzt ausgebessert, indem wir **pollFirst()** durch **takeFirst()** ersetzt haben. Wenn der Stapel des Professors leer ist, wartet er solange, bis ein Assistent wieder eine Klausur bei ihm abgelegt hat.
Um die Terminierung des Programms zu gewährleisten, mussten wir jedoch weitere Änderungen vornehmen, da es sonst zu einem Deadlock kommen kann: Der Professor kann sich ins Waitset begeben, obwohl schon alle Assistenten fertig sind und auf Terminierung warten. Da der finalstack aber nie wieder gefüllt wird, wartet der Professor endlos, und die Assistenten werden nie terminieren, da dazu der Professor **testForTermination()** aufrufen muss um sie zu interrupten und das globale Flag zu setzen. Insbesondere wird dieser Fall nicht durch die Überprüfung der while-condition verhindert, da wie bereits erwähnt der IdleAssistantsCounter nur einen Anhaltspunkt darstellt.
Deswegen führt der Professor im Falle eines leeren finalstacks Überprüfungen durch: zunächst schaut er einfach nach, ob es noch mindestens eine Klausur auf den Assistentenstapeln gibt. Ist dies der Fall, kann der Professor sich „beruhigt“ ins Wait-Set begeben, da irgendwann auf jeden Fall wieder eine Klausur bei ihm ankommen wird. Sind die Assistentenstapel jedoch alle leer, führt der Professor die umfangreichere Methode **testForTermination()** aus

(siehe 1.3) und terminiert ggf..
Dieser „zweistufige“ Ansatz soll dafür sorgen, dass die **testForTermination()** Methode möglichst selten aufgerufen wird, da sie stark sequenzialisiert und alle Assistenten anhält, während das einfache Überprüfen der Stapel recht schnell geht und die Assistenten wenig behindert.
Im Profiling des Programms war nun klar erkennbar, dass der Professor nach unserer Änderung erheblich mehr Zeit wartend verbringt als zuvor, was insgesamt auch zu etwas schnellerer Laufzeit führte.

6.3 Die Dokumentation

An unserer Dokumentation wurde kritisiert, dass sie insgesamt zu wenig Struktur aufwies und im Grunde nur ein langer Fließtext war. Diesen haben wir aufgebrochen, in dem wir eine sinnvolle Einteilung in Abschnitte vorgenommen haben: Pro Klasse ein Abschnitt (und eine neue Seite), innerhalb der Klasse falls nötig Unterabschnitte für verschiedene logische Komponenten (z.B. Initialisierungs-, Haupt- und Terminierungsphase im Professor). Dadurch und durch das beigefügte Inhaltsverzeichnis kann man zu jedem Punkt schnell die entsprechende Stelle in der Dokumentation herausfinden, ohne wie zuvor beinahe den gesamten Text lesen zu müssen.