Nebenläufige Programmierung – Praktisches Projekt
Entwurf

Vorstellung der Klassen und ihrer Methoden

Unser Entwurf sieht 4 Klassen vor: **Exam, Assistant, Professor, ExamStack**.

Der **Professor** ist hierbei die Main-Klasse und übernimmt auch die Initialisierungsphase. Darin werden abhängig von der Anzahl der Klausuren und Aufgaben zunächst Klausuren erstellt (evtl. werden die Klausuren auch bereits übergeben), diese dann auf Stapel verteilt und schließlich Assistenten-Threads erstellt. Danach werden die Assistenten gestartet und der Professor wechselt in seine Hauptphase. In dieser Hauptphase wird der Professor immer wieder zwei Aufgaben erledigen: Die Endkontrolle der Klausur und das Umverteilen der Stapel. Dazu wird er nacheinander seine Methoden **finish** und **reDistribute** aufrufen. **Finish** führt Klausuren-Endkontrolle (ggf. auch mehrere) durch und **reDistribute** stellt zunächst fest, ob sich ein Umverteilen der Klausuren lohnt und führt es dann ggf. durch. Dabei orientiert sich die Methode daran, wie viele Assistenten gerade arbeitslos sind (die kann mittels globaler Variable **waitingAssistants** geschehen, siehe unten), ob der Professor noch Klausuren zu finishen hat und wie unausgeglichen die Examenstapel sind. Die genaue Formel für ein effizientes Verteilen wird sich wohl nur durch einiges Testen ermitteln lassen. Wenn er diese Aufgaben durchgeführt hat, überprüft der Professor, ob die Korrektur schon beendet ist (siehe Unten für eine genauere Beschreibung dieses Vorgangs) und wiederholt das ganze Ansonsten. Als Examenstapel des Professors verwenden wir eine Queue, da die Assistenten nur drauf legen und der Professor nur herunter nimmt. Hierbei werden wir eine der Thread-save Queues aus dem BlockingQueue-Interface verwenden.

Der **Assistant** ist recht einfach gehalten und wird einfach eine Klausur nach der anderen kontrollieren, wobei er eine vollständig bearbeitete Klausur auf den Stapel des Professors legt und alle anderen auf den Stapel seines Nachbarn.

Die Klasse **Exam** wird erweitert durch ein boolean-Array, welches kennzeichnet, welche Aufgaben bereits kontrolliert wurden. Die Korrektur-Methode wird bei einer bereits korrigierten Aufgabe sofort returnen. Zudem bekommt das Exam eine Methode, mit der die Assistenten prüfen können, ob eine Klausur bereits vollständig korrigiert wurde.

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 wollen wir den Klausurenstapel intern in zwei Stapel aufteilen. Assistenten und Professor bekommen unterschiedliche 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. Daher muss die Klausur selber also nicht gelockt werden. 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:

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 die globale Variable **waitingAssistents** im **Professor** (mittels einer Methode **increment** zur Kapselung, die gegeneitigen Ausschluss realisiert). 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 min. ein Element hat, wird das oberste Element erst entfernt und dann zurückgegeben. 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 das Lock zurückgeben und dann auf der dazugehörigen Condition **signal()** aufgerufen.
public Exam ProfPop() 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.
public void profPush(Exam exam) Lockt den Prof-Stapel, legt das exam darauf, gibt das lock wieder frei.
private void distribute() Verteilt die Klausuren auf die Stapel. Da es nur aufgerufen wird, wenn einer der Stapel leer ist, wird es (sofern möglich) den leeren Stapel wieder aufstocken. Dabei ist es sinnvoll, mehr Klausuren auf den Prof-Stapel zu legen als auf den Assistentenstapel, da der Professor ja immer viele Klausuren hintereinander anschaut und der Assistent immer nur eine nimmt/drauflegt und danach wieder eine ganze Zeit damit beschäftigt ist, eine Aufgabe zu korrigieren.

<u>Wie der Professor merkt, dass keine Klausuren mehr da sind</u>
Es sind keine Klausuren mehr da, wenn der Professor alle Klausuren von seinem Stapel abgearbeitet hat und alle Assistenten schlafen, da sich ein Assistent nur schlafen legt, wenn er keine Klausur mehr von seinem linken Stapel nehmen kann (**condition.await** im **ExamStack**), und geweckt wird, sobald wieder etwas darauf gelegt wird (**condition.signal** im **ExamStack**).
Der Professor kann leicht überprüfen, ob sein eigener Stapel leer ist. Um zu überprüfen, ob alle Assistenten schlafen, wird er die globale Variable **waitingAssistants** auslesen. Wenn diese gleich der gesamtzahl aller Assistenten ist, bedeuted das, dass alle **condition.await** aufgerufen haben (oder als nächstes aufrufen werden, jedenfalls haben sie keine Klausur auf ihrem Stapel gefunden). Allerdings kann einer der Assistenten bereits wieder das wait set der Condition verlassen haben, aber die **decrement** methode noch nicht aufgerufen haben. Der Professor muss also überprüfen, ob nun wirklich keine Klausuren mehr da sind (**waitingAssistants** dient also nur dazu, die langwierige Prüfung seltener durchführen zu müssen).
Hierzu ruft der **Professor** auf jedem Assistenten-Thread **interrupt()** auf. Ein Assistent hat eine Schleife, in welcher er seine Arbeit erledigt, die so lange ausgeführt wird, wie er nicht interrupted wird. Wird der Assistent nun interrupted, beendet er zunächst seine aktuelle Arbeit, verlässt dann die Schleife und ruft eine Methode 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. 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 **terminated** und weckt alle Assistenten wieder auf (notifyAll()). Die Assistenten überprüfen nach dem **wait** in einer äußeren Schleife dieses boolean flag und terminieren, falls es gesetzt ist. Ansonsten arbeiten sie weiter wie zuvor.
Zur Verdeutlichung hier ein Auszug aus der run-methode des Assistenten und ein Code-Fragment aus dem Professor:

```
while(!Professor.terminate){
        while(!interrupted()){
                //do some work
                try{
                        exam = leftExam.pop()
                } catch(Interrupted exception e){
                break;
                }
                //do some work with exam
                rightExam.push(exam);
        } //before leaving this loop, we either returned the exam or we never got one.
prof.countLatch(); //counts the latch in Professor
prof.wait();
```

```
}
```

Wie der Professor prüft, ob alle Klausuren weg sind:

```
if(sleepingAssistants == numberAssistents){
        for(Thread t: assistantThreads){
                t.interrupt();
        }
        assiLatch.await();
        //teste ob alle Klausurstapel leer sind, falls ja setzte terminate auf true
        notifyAll();
}
```