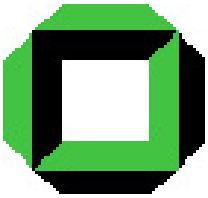

UNIVERSITÄT KARLSRUHE (TH)
Fakultät für Informatik
System Architecture Group
Frank Bellosa, Gerd Liefländer, Philipp Kupferschmied
Dominik Bruhn, Atanas Dimitrov,
Jonathan Dimond, Johannes Weiß

Basispraktikum Systemarchitektur - WS 2008/2009

Kooperation von Threads („Golfplatz“)

1 Thematik

Zwecks Zusammenarbeit nebenläufiger bzw. paralleler Threads kann man sich auf so genannte gemeinsame Daten abstützen. Man kann in diesem Fall von Kooperation sprechen, da eine vernünftige Zusammenarbeit auf gemeinsamen Daten kritisch sein kann. Während das reine Lesen gemeinsamer Information unkritisch ist, ist das Verändern von gemeinsamer Information im allgemeinen kritisch und muss demzufolge durch besondere Maßnahmen koordiniert werden. Programmabschnitte, die Zugriffe auf gemeinsame Daten beinhalten, nennt man deswegen auch kritische Abschnitte (*critical sections*). Auch gleichzeitiges Lesen und Schreiben führt u.U. zu inkonsistenter Information.

Frage 1: Geben Sie ein einfaches Beispiel an, wie gleichzeitiges Lesen und Schreiben auf gemeinsamen zu inkonsistenten Daten führen kann.

Hinweis: Machen Sie sich dabei klar, wie Anweisungen einer höheren Programmiersprache in die jeweilige Maschinensprache (bei Java in den Bytecode) übersetzt werden.

2 Grundlagen

Das gemeinsame Datum im Versuch sei ein Ganzzahlzähler, dessen Wert w, $w \in [0, max]$ (max soll zu Programmstart einstellbar sein), einem Behälter mit w Golfbällen entsprechen soll. Diese Golfbälle sollen ausleihbar sein. Man entwerfe eine beliebige Anzahl Threads (ebenso zu Programmstart einstellbar), die Bälle ausleihen und nach gewisser Zeit wieder zurückgeben.

2.1 Modellüberlegungen

Frage 2: Kann es passieren, dass die maximale Anzahl an Bällen im Behälter überschritten wird, bzw. weniger als 0 Bälle im Behälter registriert werden?

2.2 Java Monitore

Ein historisch relativ früh entwickeltes Sprachkonzept zur Zwangsserialisierung von kritischen Abschnitten sind Hoare's-Monitore. Ein Monitor entspricht einem Objekt, dessen Schnittstellenfunktionen (*member functions*) unter gegenseitigem Ausschluss stehen, d.h. so beschaffen sind, dass höchstens ein Thread eine Monitorschnittstellenfunktion ausführen kann. Mittels des Kon-

zepts der synchronisierten Klassen (*synchronized*) kann man in Java ein Monitorobjekt modellieren. Hierzu sollen Sie zwei Monitorprozeduren `addBalls()` und `removeBalls()` konstruieren, die das Ausleihen, bzw. des zurückgeben der Bälle erlaubt.

Der Randfall leerer Behälter ist abzufangen, d.h. diese Bedingung muss innerhalb einer der beiden obigen Monitorprozeduren überprüft werden. Sollte ein Thread mehr Bälle ausleihen wollen als vorhanden sind, so muss das verhindert werden, indem dieser Thread blockiert wird. Ein blockierter Thread konkurriert solange nicht mehr um den Prozessor, bis er wieder deblockiert, also bereit (*runnable*) wird. Ein Thread kann mittels der Funktion *wait()* an einer Bedingungsvariablen (*condition variable*) blockiert werden und so das Monitorobjekt verlassen, bis durch ein *notify()* bzw. *notifyAll()* in einem anderen Thread dieser blockierte Thread wieder deblockiert wird. Wie üblich garantiert Java nicht, dass der am längsten angehaltene Thread durch ein *notify()* bzw. *notifyAll()* aufgeweckt wird, es wird irgendein wartender Thread bei *notify()* deblockiert.

Eine typische Programmstelle innerhalb einer synchronisierten Monitorprozedur eines Monitorobjekts sieht wie folgt aus:

while (condition) try {wait();} catch (Interrupted Exception e) {}

Man beachte, dass an obiger Programmstelle der aufrufende Thread angehalten wird, was gleichbedeutend mit einer Threadumschaltung ist.

Hinweis: *Falls Sie Probleme mit der Java-Syntax haben, dann lesen Sie das Java-Tutorial zum Thema Threads (The Java Tutorial / Essential Java Classes / Threads: Doing Two or more Tasks at Once). Die URL lautet zur Zeit:* http://java.sun.com/docs/books/tutorial/essential/threads/

2.3 Semaphoren

Semaphoren waren das historisch erste korrekt funktionierende Synchronisationskonzept. Eine Zählsemaphore S ist ein Synchronisationsobjekt, das nach seiner Initialisierung nur über die beiden atomaren und wechselseitig-exklusiven Methoden p(S) und v(S) verwendet werden kann. Die Semantik der Semaphore wird durch folgende Ganzzahlwerte charakterisiert:

1. Ein positiver Ganzzahlwert repräsentiert die Zahl der Threads, die aktuell in ihrem kritischen Abschnitt eintreten dürfen
2. Ein negativer Ganzzahlwert repräsentiert die Zahl der Threads, die aktuell vor ihrem kritischen Abschnitt warten müssen
3. Hat die Semaphore den Wert 0, dann ist aktuell die erlaubte Anzahl von Threads in ihren kritischen Abschnitten und es wartet auch kein anderer Thread.

Die Namen der Semaphormethoden stammen von den holländischen „Passeren“ und „Verlaaten“, welche als Signale zum Einfahren in bzw. zum Verlassen der in Holland häufig anzutreffenden Schleusen dienen.
Die maximale Anzahl der Threads, die gleichzeitig in den kritischen Abschnitt eintreten dürfen, wird durch eine entsprechende Initialisierung der Semaphorenvariable festgelegt.

In Java werden die Semaphorobjekte erst im JDK Version 1.5.0 angeboten. In früheren Versionen existiert nur das sehr viel mächtigere Monitorkonzept und die Semaphoren mussten auf Basis der Monitore implementiert werden. Implementieren und testen Sie Ihre Semaphoren zunächst getrennt.

Hinweis: Bei der Lösung der Aufgaben dürfen Sie die in jdk 1.5 angebotene Implementierung des Semaphorenkonzepts nicht verwenden. Sie müssen die Semaphoren selbst entwickeln.

3 Experimente

3.1 Einmal spielen bitte ...

Die Java-Applikation beinhalte als Aufrufparameter die initiale Zahl von Golfspielern, sowie die Anzahl der ausleihbaren Bälle. Die Golfspieler sollen zwischen 1 bis 5 Bälle einmalig ausleihen und sich nach Rückgabe beenden (wer mehr Bälle ausleiht, spielt natürlich auch länger – schließlich braucht er mehr Zeit alle Bälle wiederzufinden !).
Realisieren Sie die Koordination der Ballausleihe einmal mit Java-Monitoren und einmal mit Hilfe Ihrer Semaphoren.

Hinweis: Die Idee, Semaphoren direkt als Ballbehälter zu verwenden, kann zu Verklemmungen (deadlocks) führen.

Frage: Welche Unterschiede und Gemeinsamkeiten haben die beiden Varianten und warum?

Frage: Welches Problem entsteht bei der Verwendung von Semaphoren für den Fall, dass nicht mehr genügend Bälle vorhanden sind ? Finden Sie eine Lösung ? Ist diese Behandlung in der anderen Variante mit Monitoren einfacher zu realisieren ?

3.2 ... aber es war doch so schön!

Man erweitere obiges Programm so, dass die Threads beliebig oft Bälle ausleihen können (allerdings auch eine kurze Pause einlegen vor dem nächsten Spiel !). Es sind wieder beide Varianten wie im obigen Versuch zu implementieren.

Wichtig: Achten Sie darauf, dass die korrekte Funktionsweise Ihres Programms sowohl aus dem Code als auch aus der Textausgabe ersichtlich ist.

3.3 Visualisierung und Ereignisknöpfe für den Ballbehälter.

Denken Sie sich eine geeignete Visualisierung des Status der Golfspieler und des Ballbehälters aus. Man könnte beispielsweise als graphischen Pegel den jeweiligen Füllstand des Ballbehälters visualisieren. Unter Verwendung des Programms aus 3.2 realisiere man ein Beobachtungsfenster mit zwei Knöpfen um die Anzahl der Bälle im Behälter (und somit auch die Grenze für die maximalen Anzahl von Bällen) zu verändern. Unmögliche Ereignisse müssen abgefangen werden!