Informatik B

Vorlesung 8

Synchronisierung von Threads

Rückblick

- Threads
- Erzeugen, Starten und Stoppen
- Fehler abfangen
- Gemeinsamer Zugriff auf Ressourcen
- Kritische Blöcke (Einleitung)

Kritische Blöcke

- Kritische Stellen im Programmcode müssen vor paralleler Bearbeitung geschützt werden
- Beispiel: Ein Thread iteriert über ein Array, ein anderer sortiert das Array gleichzeitig
- Dazu muss ein solcher Bereich gesperrt werden
- Es gibt in Java zwei Konzepte zur Absicherung:
  - Synchronisieren
  - Lock-Objekte
- Tritt ein Thread in einen kritischen Bereich ein, wird dieser gesperrt
- Nachfolgende Threads müssen warten, bis die Sperrung wieder freigegeben ist

`synchronized`

- Das Schlüsselwort **`synchronized`** leitet einen geschützten Bereich ein, bzw. definiert eine ganze Methode als **`synchronized`**
- **`synchronized`** schützt immer Blockweise
- Um einen Block zu schützen bedarf es eines Monitor-Objektes
- Das Monitor-Objekt sollte allen Threads bekannt und eindeutig sein
- Wird ein **`synchronized`** Block betreten wird dies am entsprechenden Monitor gekennzeichnet
- Ist der kritische Block beendet wird das Monitor-Objekt wieder freigegeben

synchronized

- Soll eine ganze Methode als kritisch eingestuft werden kann folgendes geschrieben werden:
  `synchronized void m1() { // Anweisungen }`
- In diesem Fall ist das Monitor-Objekt **`this`**
- Ein Sperren einer gesamten Methode ist aber oftmals nicht sinnvoll
- Es sollten immer nur feingranulare kritische Blöcke definiert werden

```
public void m2() {
    // Nicht kritische Anweisungen
    synchronized {
          // Kritische Anweisungen
    }
    // Nicht kritische Anweisungen
}
```

`synchronized`

- Eine Synchronisation auf **`this`** ist nur dann ratsam, wenn die Manipulationen ausschließlich auf Variablen von **`this`** durchgeführt werden
- Werden Variablen von anderen referenzierten Objekten verändert, ist ein synchronisieren auf **`this`** unnötig weitgreifend
- Daher sollte ein anderes Monitor-Objekt gewählt werden
- Dieses Monitor-Objekt wird in Klammern hinter das Schlüsselwort **`synchronized`** geschrieben:
  `synchronized(monitorObject) { // Kritische Anweisungen }`
- Beispiel: **`synchronize1`**

Monitor-Objekt

- Die Wahl des Monitor-Objektes ist nicht immer einfach
- Man muss genau schauen, welche Objekte innerhalb eines kritischen Blockes manipuliert werden
- Man muss außerdem darauf achten, dass der Monitor für alle Threads gleich ist
- Beliebter Fehler ist die Synchronisation auf this innerhalb der **`run()`**-Methode eines Threads
- Dies kann nicht funktionieren, da jeder Thread auf sich selbst synchronisieren würde
- Beispiel: **`synchonize2`**

Monitor-Objekt

- Am besten ist es das Objekt als Monitor zu verwenden, welches manipuliert wird
- Werden mehrere Objekte manipuliert kann man z.B. ein Klassen-Objekt (welches immer nur einmal existiert) als Monitor verwenden
- Nachteil dieser Lösung ist, dass auch andere Blöcke ggf. dieses Objekt als Monitor verwenden
- Dadurch können Bereiche gesperrt sein, die im Moment ohne Probleme durchgeführt werden könnten
- Eine andere Alternative sind daher eigens eingeführte Klassenvariablen, die nur als Monitor verwendet werden
- Ein solches Objekt sollte final sein
- Dadurch hätte der Programmierer ein bestimmtes Objekt, welches nur für seine Monitor-Zwecke vorhanden wäre
- Beispiel: **`synchronize3`**

Reentrant

- Betritt ein Thread einen synchronisierten Block, bekommt es den gesetzten Monitor
- Wenn der Thread eine andere Methode aufruft, in der ein Block vorhanden ist, der am gleichen Objekt synchronisiert ist, kann der Thread sofort eintreten und muss nicht warten
- Diese Eigenschaft heißt *reentrant*
- Ohne diese Möglichkeit würde z.B. Rekursion nicht funktionieren!

Locks

- Seit Java5 gibt es ein weiteres Konzept, um kritische Bereiche vor dem gemeinsamen Zugriff mehrerer Threads zu schützen
- Dazu dienen Klassen, die das Interface **`java.util.concurrent.locks.Lock`** implementieren
- Insgesamt implementieren drei Klassen das Interface:
  - **`ReentrantLock`**
  - **`ReadLock`**
  - **`WriteLock`**

Locks

- Ein Block kann mittels eines **`Lock`**-Objektes mit der Methode **`lock()`** gesperrt und mit der Methode **`unlock()`** wieder freigegeben werden

```
Lock lock = …;
lock.lock();
// kritischer Bereich
lock.unlock();
```

- Hierbei ist es wichtig darauf zu achten, dass die Sperrung auch wirklich aufgehoben wird
- Vergisst der Programmierer den Aufruf von **`unlock()`** bleibt das Programm ggf. „hängen“
- Ebenso verhält es sich bei auftretenden Exceptions
- Wird eine Exception vor dem **`unlock()`** Aufruf geworfen und die Anweisung nicht mehr abgearbeitet wird der Lock nicht freigegeben
- Abhilfe schafft hierbei die Verwendung von **`try/finally`**-Konstrukten
- Beispiel: **`lock1`**

Locks

- Die Verwendung von Locks ist gegenüber der Verwendung des **`synchronized`** nicht auf Blöcke begrenzt
- Locks erlauben dadurch eine flexiblere Programmierung
- Beispiel: **`locks2`**
- Allerdings sind Locks schwieriger zu handhaben und es können sich schneller Fehler einschleichen
- Daher lohnt es sich zunächst **`synchronized`** zu verwenden und nur wenn dies nicht klappt auf Locks zurückzugreifen

Warum nicht alles synchronisieren?

- In nebenläufigen Programmen kann es schnell zu unerwünschten Nebeneffekten kommen
- Das ist auch der Grund, warum threadlastige Programme schwer zu debuggen sind
- Warum sollten man also nicht alle Methoden synchronisieren?
- Methoden, die synchronisiert sind, müssen von der JVM besonders bedacht werden, damit keine zwei Threads die Methode für das gleiche Objekt ausführen
- Das kostet zusätzlich Zeit und ist im Vergleich zu einem normalen Methodenaufruf teurer
- Zusätzlich kommt ein Problem hinzu, wenn eine nicht notwendigerweise, also überflüssige, synchronisierte Methode eine Endlosschleife oder lange Operationen durchführt
- Dann warten alle anderen Threads auf die Freigabe, und das kann im Fall der Endlosschleife ewig sein
- Wenn alle Methoden synchronisiert sind, steigt auch die Gefahr eines unnötigen Deadlocks

Deadlocks

- Verhindert man die gleichzeitige Abarbeitung einzelner Programmbereiche können Deadlocks entstehen
- Ein Deadlock liegt vor, wenn zwei Threads aufeinander warten
- Beispiel: **`deadlock1`**

Quelle:
„Java ist auch eine Insel“ von Christian Ullenboom
Programmieren mit der Java Standard Edition Version 6

Deadlocks

- Wenn das Programm nicht mehr reagiert kann man auf der Unix/Linux Konsole oder auf der Kommandozeile unter Windows die Tastenkobination Strg+Pause drücken
- Man erhält dann eine komplette Übersicht aller Threads
- Im Idealfall erkennt die JVM vorhandene Deadlocks und meldet diese
- Zudem gibt es die **`jconsole`**
- Dabei handelt es sich um eine grafische Anwendung, mit der ein Javaprogramm zur Laufzeit betrachtet werden kann
- Dazu muss das Javaprogramm mit einer Property gestartet werden, die anzeigt, dass die **`jconsole`** sich an das Javaprogramm hängen darf:
  `java -Dcom.sun.management.jmxremote deadlock1.Test`
- Danach kann **`jconsole`** aufgerufen werden und Informationen zum laufenden Javaprogramm werden angezeigt
- Demo

Deadlocks vermeiden

- Eine Möglichkeit Deadlocks zu vermeiden ist immer in der gleichen Reihenfolge Monitore oder Locks zu sperren
- Allerdings führt auch das nicht immer zum Ziel
- Abhilfe schaffen bei **`synchronized`** Blöcken die Methoden **`wait()`** und **`notify()`**
- Bei Locks helfen die Methoden **`await()`** und **`signal()`** weiter

`wait()` und `notify()`

- Um mit **`wait()`** und **`notify()`** zu arbeiten, müssen die Methoden an dem Monitor-Objekt des **`synchronized`** Blockes aufgerufen werden
- Beispiel:
  - Seien zwei Threads gegeben
  - Sie synchronisieren sich am Objekt o
  - Die Methoden **`wait()`** und **`notify()`** sind nur mit dem entsprechenden Monitor gültig, und den besitzt das Programmstück, wenn es sich in einem synchronisierten Block aufhält
  - Thread **`T1`** soll auf Daten warten, die Thread **`T2`** liefert

```
synchronized(o) {
  try {
          o.wait();
          // Habe gewartet, kann jetzt loslegen
  } catch ( InterruptedException e ) {
          ... }
}
```

`wait()` und `notify()`

- Beispiel (Fortsetzung)
  - Wenn der zweite Thread den Monitor des Objekts **`o`** bekommt, kann er den wartenden Thread aufwecken
  - Der zweite Thread kann nur den Monitor bekommen, wenn er nicht bereits belegt ist
  - Der Aufruf von **`wait()`** von **`T1`** gibt den Monitor wieder frei
  - **`T2`** kann in den **`synchronized`** Block eintreten, Aktionen durchführen und dann **`T1`** wieder wecken
  - **`T2`** gibt das Signal mit **`notify()`**

```
synchronized( o ) {
  // Habe etwas gemacht und informiere jetzt
  // meinen Wartenden
  o.notify();
}
```

`wait()` und `notify()`

- **`wait()`** stellt den aktiven Thread in eine interne Warteschlange des Monitors, bis ein anderer Thread die Methode **`notify()`** des Monitors aufruft
- Eine zweite Variante von **`wait()`** erlaubt die Angabe einer Zeitspanne, die maximal gewartet werden soll. Falls das Ereignis innerhalb dieser Zeit nicht eintritt, setzt der Thread seine Ausführung fort
- Wird innerhalb eines **`synchronized`** Blockes mit dem Monitor **`o`** die Methode **`wait()`** aufgerufen, kann dieser Thread nur durch einen anderen Thread mit einem **`notify()`** geweckt werden, wobei dieser Aufruf ebenfalls innerhalb eines **`synchronized`** Blockes stehen muss, der auf das gleiche Monitor-Objekt **`o`** synchronisiert
- Dies klingt zunächst etwas verwirrend, da ja immer nur ein Thread gleichzeitig in einer solchen Methode aktiv sein kann
- Dies klappt, da **`wait()`** die Eigenschaft hat, den Monitor freizugeben, so dass dieses Problem nicht auftritt

`wait()` und `notify()`

- **`notify()`** bewirkt, dass ein beliebiger der wartenden Threads aufgeweckt und die Ausführung fortsetzt
- Welcher das ist, darüber macht die Sprachspezifkation keine Vorgaben
- Für Anwendungsfälle, in denen nicht nur einer, sondern alle wartenden Threads vom Eintritt des Ereignisses benachrichtigt werden sollen, definiert **`Object`** die Methode **`notifyAll()`**
- Es werden dann alle wartenden Threads geweckt, wobei der Scheduler entscheidet, welcher Thread als erster drankommt

Consumer-Producer

- Das vorangegangene Beispiel wird als Consumer-Producer-Problem bezeichnet
- Ein Thread produziert etwas (Producer), auf das der andere Thread (Consumer) warten muss, und stellt es in einen Zwischenpuffer
- Ggf. muss auch der Producer Thread warten, falls der Puffer voll ist und der Consumer erst die Daten abholen muss
- Beispiel: **`producerconsumer1`**

Many Consumer - many Producer

- Das Beispiel mit **`wait()`** und **`notify()`** ist unproblematisch, solange ein Producer und ein Consumer vorhanden ist
- Mit **`notify()`** wird immer genau der andere benachrichtig, so dass nach einem Producer ein Consumer an der Reihe ist und umgekehrt
- Stellen wir uns beliebig viele Consumer und Producer vor
- Arbeitet einer der beiden Threadtypen auf dem Puffer gibt er mit **`notify()`** einem beliebigen Thread das Zeichen weiterzuarbeiten
- Hat z.B. ein Producer den Puffer gefüllt benachrichtigt er einen anderen Thread
- Der Producer könnte aber einen weiteren Producer wecken, der eigentlich nicht arbeiten kann, da der Puffer voll ist
- Beispiel: **`producerconsumer2`**

Many Consumer – many Producer

- Das Problem kann mit einer **`while`**-Schleife um den **`wait()`**-Aufruf gelöst werden
- Wird ein Thread also geweckt und beginnt mit der Arbeit schaut er erst nach, ob die Ressource verfügbar ist
- Beispiel: **`producerconsumer3`**
- Das führt zu einem weiteren Problem
- **`notify()`** weckt immer einen beliebigen Thread
- Z.B. warten alle Producer auf den freiwerdenden Puffer
- Ein Consumer verbraucht ein Element und ein Producer wird geweckt und erzeugt ein Element für den Puffer
- Dann sagt der Producer **`notify()`** und durch Zufall wird ein Producer geweckt, dieser hängt aber in der Schleife fest, da der Puffer voll ist
- Niemand sagt mehr **`notify()`** => das Programm hängt

Many Consumer – many Producer

- Abhilfe schafft hier die Methode **`notifyAll()`** die alle Threads weckt
- So können sich alle Threads wieder um den Puffer schlagen und irgendeiner kann arbeiten
- Generell ist diese Vorgehensweise bei der Programmierung sinnvoll, da man nie weiß wie viele Threads wirklich erzeugt werden
- Beispiel: **`producerconsumer4`**

Busy wait

- Anstelle des **`wait()`**-Aufrufes könnte man auf den Gedanken kommen einen Thread in einer **`while`**-Schleife **ohne** **`synchronized`** Block und **`wait()`** auf die Freigabe von Ressourcen warten zu lassen
- Dies ist aber nicht sinnvoll, da dann immer wieder abgefragt wird, ob weitergemacht werden kann oder nicht
- Dieser Vorgang wäre sehr Rechenintensiv und nennt sich entsprechend *Busy wait*
- Beispiel: **`producerconsumerbusywait`**

`await()` und `signal()`

- Analog zu **`wait()`** und **`notify()`** bei **`synchronized`** Blöcken werden bei Locks die Methoden **`await()`** und **`signal()`** verwendet
- Diese Methoden werden an einem **`Condition`**-Objekt aufgerufen
- Das **`Condition`**-Objekt wird von der Methode **`newCondition()`** einer Lock-Klasse zurückgeliefert

```
Condition condition = lock.newCondition();
```

- Thread **`T1`** soll auf Daten warten, die Thread **`T2`** liefert

```
try {
    condition.await();
} catch ( InterruptedException e ) {
    ...
}
```

- Mit der Methode **`await()`** geht der Thread in den Zustand „nicht ausführend“ über

`await()` und `signal()`

- Um einen anderen Thread aufzuwecken wird die Methode **`signal()`** verwendet:
  `condition.signal();`
- Sollen alle wartenden Threads einen Hinweis bekommen wird die Methode **`signalAll()`** verwendet
- Die Mehoden **`await(), signal()`** und **`signalAll()`** können nur innerhalb eines gesperrten Bereiches aufgerufen werden
- Werden die Methoden außerhalb eines gesperrten Bereiches aufgerufen wird eine **`java.lang.IllegalMonitorStateException`** geworfen
- Beispiel: **`producerconsumer5`**

Rückblick

- **`synchronized`**
- Locks
- Deadlocks
- **`wait()`** und **`notify()`**
- **`await()`** und **`signal()`**
- Producer/Konsumer

Ausblick

- Semaphore
- Volatile
- ThreadGroups

- Collection-Framework
- Generische Typen