Informatik B

Vorlesung 9

Deadlocks, Semaphor, ThreadGroups

Rückblick

- **`synchronized`**
- Locks
- Deadlocks
- Producer/Consumer

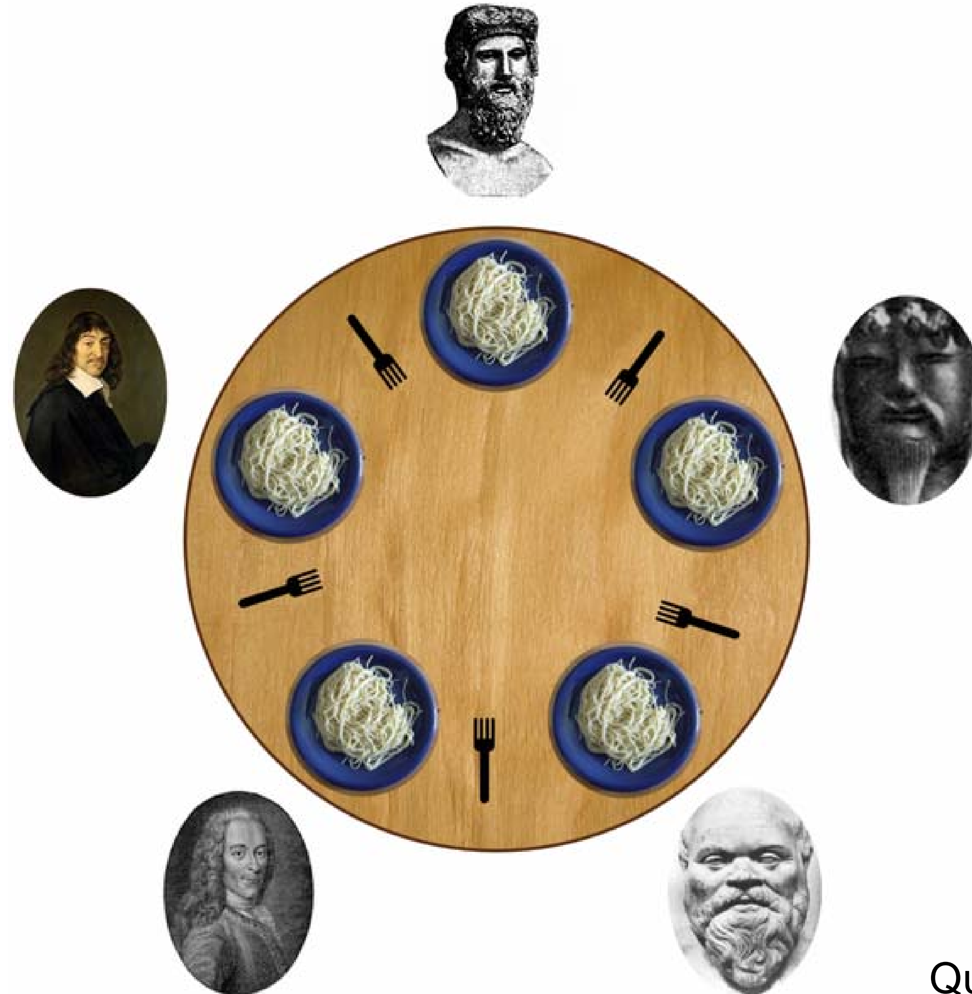
Producer/Consumer

- Wiederholung
- Beispiel: **`producerconsumer1`**

Dining Philosphers

- Ein klassisches Beispiel für eine Deadlocksituation sind die *Dining Philosphers*
- An einem Runden Tisch sitzen **`n`** Philosophen
- Zwischen jeweils zwei Philosophen befindet sich ein Essstäbchen
- Ein Philosoph spricht nicht, sondern denkt nur nach
- Irgendwann bekommt ein Philosoph Hunger und greift nach dem rechten Stäbchen
- Hat ein Philosoph das rechte Stäbchen versucht er nach dem linken zu greifen, ggf. wartet er bis das Stäbchen frei ist
- Hat ein Philosoph beide Stäbchen kann er essen
- Nachdem er gegessen hat, legt er die Stäbchen hin

Dining Philosphers



Quelle: Wikipedia

Dining Philosphers

- Klasse **`ChopStick`**
  - Synchronisierte **`get()`** und **`put()`** Methoden zum Ergreifen und Loslassen eines Stäbchen
  - Soll ein nicht freies Stäbchen aufgenommen werden, wird gewartet, bis dieses frei ist
  - Wird ein Stäbchen abgelegt, wird **`notifyAll()`** an alle wartenden Philosophen gesagt

Dining Philosphers

- Klasse **`Philosopher`**:
  - Ein Philospoh lebt in einer Endlosschleife
  - Er denkt nach (**`sleep()`**)
  - Nimmt das rechte Stäbchen
  - Wartet ggf. auf die Freigabe
  - Nimmt das linke Stäbchen
  - Wartet ggf. auf die Freigabe
  - Isst eine gewisse Zeit (**`sleep()`**)
  - Er gibt beide Gabeln wieder frei

Dining Philosphers

- Hauptprogramm
  - Es werden einige Philosophen und entsprechend viele Stäbchen angelegt und positioniert
  - Die Philosophen-Threads werden gestartet
- Beispiel: **`diningphilosophers1`**

Dining Philosphers

- Problem:
  - Es tritt recht wahrscheinlich ein Deadlock auf (alle haben das rechte Stäbchen)
  - Wenn alle warten kann niemand mehr `notifyAll()` sagen, so dass kein Philosoph mehr aufwacht
- Lösung des Problems:
  - Ein Philosoph handelt anders. Er nimmt erst das linke und dann das rechte => keine Deadlocks mehr
  - Ein Philosoph müsste nachgeben, wenn er das erste Stäbchen besitzt, das zweite aber zur Zeit nicht erhalten kann
  - Beide Lösungen sind nicht im eigentlichen Sinne des Problems

Dining Philosophers

- Annahme:
  - Der erste Philosoph handelt anders und nimmt zuerst das linke und dann das rechte Stäbchen
- Dadurch wird er selber und sein linker Nachbar weniger zu essen bekommen, als die anderen
- Der rechte Nachbar wird am meisten zu essen bekommen, da er bevorteilt wird
- => Diese Lösung funktioniert zwar, sorgt aber nicht für eine gleichberechtigte Verteilung der Ressourcen
- Beispiel: **`diningphilosophers2`**

Dining Philosophers

- Annahme:
  - Ein Philosoph nimmt das erste Stäbchen und testet dann ob das zweite verfügbar ist
  - Ist dies der Fall isst er, ansonsten legt er beide Stäbchen zurück
- Dadurch werden belegte Ressourcen immer wieder freigegeben
- Es wird beim zweiten Stäbchen nicht gewartet, sondern nur beim ersten
- Diese Lösung funktioniert, entspricht aber nicht genau der Aufgabenstellung
- Beispiel: **`diningphilosophers3`**

Semaphore

- Ein Semaphor lässt nur eine bestimmte Anzahl von Threads auf ein Programmstück zugreifen
- Es lassen sich zwei Typen von Semaphoren unterscheiden:
  - Binäre Semaphore lassen höchstens einen Thread auf ein Programmstück zu.
  - Allgemeine Semaphore lassen eine bestimmte begrenzte Menge an Threads in einen kritischen Abschnitt

Semaphore

- Ein Semaphor verwaltet intern eine Menge so genannter Erlaubnisse (eng. *permits*)
- Das bekannte Paar **`await()/singal()`** beziehungsweise **`wait()/notify()`** bietet sich für einen binären Semaphor an
- Für allgemeine Semaphoren mit einer maximalen Anzahl Threads im Programmstück deklariert die Java-Bibliothek die Klasse **`java.util.concurrent.Semaphore`**
- Auch ein binärer Semaphor kann mit der Klasse **`Semaphore`** implementiert werden

Semaphore

- Intern vermerkt ein Semaphor das Betreten von einem Block eines Threads
- Ein Semaphor lässt Threads warten, wenn das gesetzte Maximum erreicht ist, bis ein anderer Thread das Programmsegment verlässt
  - **`Semaphore(int permits)`** Ein neuer Semaphor, mit der Angabe wie viele Threads in einem Block sein dürfen
  - **`void acquire()`** Versucht, in den kritischen Block einzutreten; wenn der gerade belegt ist, wird gewartet; vermindert die Menge der Erlaubnisse um eins
  - **`void tryAcquire()`** Versucht, in den kritischen Block einzutreten; wenn der gerade belegt ist, wird sofort **`false`** zurückgeliefert, ansonsten wird die Menge der Erlaubnisse um eins vermindert
  - **`void release()`** Verlässt den kritischen Abschnitt und legt eine Erlaubnis zurück
- Beispiel: **`semaphore1`**

Semaphore

- Falls ein Block aufgrund eines freien Platzes im Semaphor betreten werden darf, wählt die Methode **`acquire()`** einen beliebigen wartenden Thread aus, der den Block betreten darf
- Das hat unter Umständen zur Folge, dass ein Thread seltener als ein anderer den Block betreten darf
- Abhilfe schafft ein fairer Semaphor der mit folgendem Konstruktor erzeugt werden kann:
  `new Semaphore(int permits, boolean fair)`
- Wird der Parameter **`fair`** auf **`true`** gesetzt kommt jeder Thread gleich oft an die Reihe
- Beispiel: **`semaphore2`**

Dining Philosophers

- Das *Dining Philosophers* Problem kann recht einfach mit binären Semaphoren implementiert werden
- Eine Implementation, bei der die Philosophen die Ressource Essstäbchen ggf. wieder freigeben findet sich im Beispiel: **`diningphilosphers4`**

Sleeping Barber

- Ein klassisches Problem für die Verwendung von Semaphoren ist das *Sleeping Barber Problem:*
  - Ein Frisör hat einen Frisierstuhl und eine bestimmte Anzahl an Stühlen für wartende Kunden
  - Sind keine Kunden da, schläft der Frisör
  - Kommt ein Kunde in das Geschäft schaut er sich um und nimmt auf einen Wartestuhl Platz, insofern einer frei ist. Ist kein Stuhl frei geht er wieder und kommt nach einiger Zeit wieder
  - Ein wartender Kunde schaut nach, ob der Frisierstuhl frei ist und setzt sich dorthin falls der Stuhl frei ist und weckt den Frisör
  - Der Frisör schneidet dem Kunden die Haare und der Kunde macht danach den Frisierstuhl wieder frei
  - Jeder Kunde kommt nach einem gewissen Zeitintervall wieder

Sleeping Barber

- Bei diesem Problem gibt es verschiedene Punkte zu beachten:
  - Nur ein Kunde kann gleichzeitig die Haare geschnitten bekommen
  - Es können nur eine begrenzte Anzahl von Kunden warten
  - Die Zugriffe auf die Ressourcen stellen kritische Abschnitte dar
  - Der *Sleeping Barber* sollte möglichst wenig Ressourcen verbrauchen
  - Es werden mehrere Sperrmechanismen benötigt, um einen reibungslosen Ablauf zu garantieren

Sleeping Barber

- Der Barber ist ein Thread, der nur schläft
- Da ein `sleep()` nicht unendlich dauert, kann der Barber theoretisch während des `sleep()` geweckt werden oder im kurzen Bereich bis zum neuen `sleep()` Aufruf
- Ab und an wird er geweckt
- Dies führt entweder zu einer `InterruptedException` oder zu einem Statuswechsel des Threads (insofern der `interrupt()` Aufruf zwischen zwei `sleep()` auftrat)
- Sobald er geweckt wurde, schneidet er einem Kunden die Haare
- Beispiel: `sleepingbarber.SleepingBarber.java`

Customer

- Der Customer durchläuft mehrere Phasen
- Als erstes muss er einen Platz im Geschäft bekommen. Dazu muss er den Semaphor für die freien Plätze befragen
- Ist kein Platz frei verlässt der Customer das Geschäft und wartet eine bestimmte Zeit
- Falls ein Platz frei ist belegt der Customer einen Platz und muss das bei dem entsprechenden Semaphor vermerken
- Dann wartet er auf den Semaphor der den einen freien Platz des Barber verwaltet
- Hat er den freien Platz ergattert gibt er bei dem Semaphor für die Warteplätze wieder einen frei
- Dann schickt er dem Barber eine Methode zum Wecken und übergibt eine Referenz von sich selbst
- Wurde dem Customer die Haare geschnitten geht er aus dem Geschäft, wartet und will dann erneut die Haare geschnitten haben
- Beispiel: **`sleepingbarber.Customer.java`**

HauptProgramm

- Entsprechend der Anzahl der Warteplätze wird ein `Semaphore`–Objekt initialisiert
- Ein `Semaphore`–Objekt für den Barber wird angelegt
- Die Customer werden erzeugt (mit den beiden Semaphoren und dem Barber) und gestartet
- Zusätzlich wird ein `TimerTask` erzeugt, der alle zwei Sekunden den Status aller Customer ausgibt
- Beispiel: `sleepingbarber.RealWorld.java`

`volatile`

- Die JVM arbeitet bei den Ganzzahl-Datentypen kleiner gleich `int` intern nur mit einem `int`
- Die Zugriffe auf ein `int` werden in einem Bytecode-Befehl abgearbeitet
- Bei den 64-Bit-Datentypen `long` und `double` gibt es zwei Bytecode-Befehle jeweils für die 32 Bit-Zuweisung
- Daher ist nicht gesichert, dass die Operationen auf diesen Datentypen unteilbar sind

`volatile`

- Es kann also passieren, dass ein Thread mitten in einer `long`- oder `double`-Operation von einem anderen Thread verdrängt wird
- Greifen zwei Threads auf die gleiche 64-Bit-Variable zu, so könnte möglicherweise der eine Thread eine Hälfte schreiben und der andere Thread die andere
- Um dies zu vermeiden, können die Objekt- und Klassenvariablen mit dem Schlüsselwort `volatile` deklariert werden

`volatile`

- Die Zugriffsoperationen auf Variablen mit dem Schlüsselwort `volatile` werden dann atomar ausgeführt
- Das Schlüsselwort ist bei lokalen Variablen nicht gestattet, da sie für andere Threads nicht zugänglich sind
- Achtung: Auch mit `volatile` sind Operationen wie `i++` natürlich nicht atomar

ThreadGroup

- Jeder erzeugte Thread gehört zu einer Gruppe, die durch ein `ThreadGroup`-Objekt repräsentiert wird
- Wird keine `ThreadGroup` angegeben wird der neu erzeugte Thread zu der Gruppe des erzeugenden Threads hinzugefügt
- Eine explizite Zuordnung zu einer `ThreadGroup` ist ebenfalls möglich

ThreadGroup

- Die **ThreadGroupS** sind Baumartig angeordnet
- Eine **ThreadGroup** kann also weitere enthalten
- Die Wurzel der Benutzerthreads bildet die Gruppe **main**
- Die Wurzel aller Threads bildet die Gruppe **system**
- Beispiel: **threadgroup1**

ThreadGroup

- Sollen Threads zu einer bestimmten **`ThreadGroup`** hinzugefügt werden, muss dafür zunächst ein **`ThreadGroup`**-Objekt erzeugt werden:
  `new ThreadGroup(„Name“);`
- Threads können dann zu dieser Gruppe hinzugefügt werden:

```
new Thread(group, name);
new Thread(group, runnable, name);
```

- Beispiel: **`threadGroup2`**

ThreadGroup

- Vorteil einer **`ThreadGroup`** ist die bessere Verwaltung mehrer Threads
- Mit einer **`ThreadGroup`** können so mehrere Threads gleichzeitig unterbrochen werden, als Dämon klassifiziert werden, usw.

Threads: Zusammenfassung

- Threads mittels `Thread`-Objekten und `Runnable`-Objekten erstellen
- Threads starten
- Threads unterbrechen
- Threads schlafen legen
- Auf Threads warten (`join()`)

Threads: Zusammenfassung

- Synchronisieren
  - **`synchronized`**, Monitor-Objekt
  - **`lock()/unlock()`**, Lock-Objekt
- Deadlocks
- Producer/Consumer
- Dining Philosophers
- **`wait()/notify()/notifyAll()`**
- **`await()/signal(),signalAll()`**
- **`Semaphore`**
- Sleeping barber
- **`ThreadGroup`**

Ausblick

- Collection Framework