Ohne Locks geht's nicht, oder?

Paradigmen der nebenläufigen Programmierung – Teil 1: atomic und @Immutable

Johannes Link

Ein Problem der Programmierung nebenläufiger Prozesse ist der Zugriff auf gemeinsame Ressourcen. Das gängige Programmiermodell für die nebenläufige Programmierung unter Java setzt auf explizite Synchronisation. Die Feinheiten dieses Ansatzes sind schwer zu erlernen und in komplexeren Systemen kaum fehlerfrei zu implementieren. Es existieren jedoch eine Reihe alternativer Programmierparadigmen, die versprechen, den Umgang mit nebenläufigen Problemstellungen einfacher zu machen. In diesem zweiteiligen Artikel werden einige dieser Ansätze vorgestellt und mit Beispielen in Java und dem Groovy-Framework GPars erläutert. Im ersten Teil geht es um Alternativen zu den Locks, im zweiten Teil um die Ausnutzung des Parallelisierungspotenzials.

Javas Locking-Ansatz

Vor der Einführung von Java 5 waren die Bordmittel des JDK zur Unterstützung nebenläufiger Programmierung an einer Hand abzählbar: Mit den Schlüsselwörtern **synchronized** und **volatile** sowie den Methoden **wait()**, **notify()** and **notifyAll()** musste man alle Probleme der Synchronisation und des bedingten Wartens erledigen. Mit Java 5 kamen eine Reihe zusätzlicher Hilfsmittel hinzu, die dem Java-Entwickler das parallele Leben einfacher machen. Dazu gehören u. a. eigene Lock-Klassen, welche die Möglichkeiten des klassischen **synchronized** abdecken und erweitern. Auch bietet das JDK nun mit Atomic*-Variablen die Basis für eine lock-freie Nebenläufigkeit bereits von Haus aus an. Einen Überblick über Javas aktuelle Werkzeuge für die Programmierung mit Threads gibt [Hent10] in diesem Heft.

Trotz aller neuen Features in **java.util.concurrent** bleibt die Grundidee die gleiche: Wir verwalten unseren gemeinsamen Zustand (*shared state*) in Objekten und sorgen (meist) mithilfe von Locks dafür, dass sich gleichzeitig ablaufende Threads beim Zugriff auf diese Objekte nicht in die Quere kommen. Sind wir beim Einsatz – oder beim Vermeiden – dieser Locks nur geschickt genug, dann erreichen wir die drei Hauptziele nebenläufiger Programmierung: Safety, Liveness und Performance.

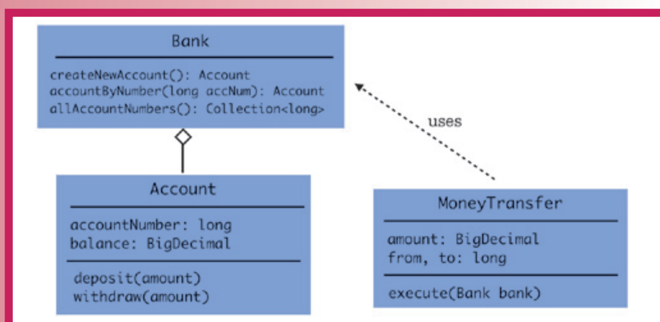


Abb. 1: Bank-Konto-Überweisung

Schauen wir uns die Programmierpraxis an dem in Abbildung 1 skizzierten Beispiel an. Das fachliche Problem ist einfach und vertraut: Wir können an einem **Bank**-Objekt Konten erzeugen, die eine eindeutige Kontonummer erhalten. Neben Ein- und Auszahlungen auf die Konten wollen wir Überweisungen von einem auf ein anderes Konto vornehmen können, falls der entsprechende Betrag verfügbar ist. Alle Operationen sollen in parallelen Threads sicher ausführbar sein. Betrachten wir zunächst die Implementierung der Bank:

```
public class Bank {
  private final Map<Long, Account> accounts =
    new ConcurrentHashMap<Long, Account>();
  private final AtomicLong lastAccountNumber =
    new AtomicLong(0L);
  public Account createNewAccount() {
    Account account =
      new Account(lastAccountNumber.incrementAndGet());
    accounts.put(account.getAccountNumber(), account);
    return account;
  }
  public Account accountByNumber(long accountNumber) {
    return accounts.get(accountNumber);
  }
}
```

Durch die Wahl einer **ConcurrentHashMap** für die Konten und eines **AtomicLong** für die Kontonummern kommen wir glücklicherweise ganz ohne explizite Synchronisation aus. „Lock free concurrency“ (s. auch [Hent10]) ist meist performanter als lock-basierte Synchronisation, dafür aber nicht so universell einsetzbar. Durch den **final**-Modifizierer an unseren Member-Variablen müssen wir zusätzlich dafür sorgen, dass wir den gemeinsam verwendeten Zustand auch für alle Threads sichtbar publizieren. Diesen Vorgang nennt man *Safe Publishing* und er resultiert aus Javas Speichermodell, das nur sehr vage Zusicherungen für veränderliche, nicht-synchronisierte Zustandsvariablen macht.

Auch die **Account**-Klasse könnten wir durch die Verwendung einer **AtomicReference**-Variablen für den Saldo lock-frei und dennoch thread-sicher gestalten. Das sähe dann etwa so aus:

```
public class Account...
  private final AtomicReference<BigDecimal> balance =
    new AtomicReference<BigDecimal>(BigDecimal.ZERO);
  public void deposit(BigDecimal amount) {
    while (true) {
      BigDecimal oldAmount = balance.get();
      BigDecimal newAmount = oldAmount.add(amount);
      if (balance.compareAndSet(oldAmount, newAmount))
        return;
    }
  }
```

Wir vermeiden hier eine klassische *Race Condition*. So nennt man den Fall, wenn es bei einer bestimmten nebenläufigen Konstellation – ein Thread überholt beim Rennen einen anderen – zu Inkonsistenzen in der Fachlogik kommen kann. Bei einer nicht-atomaren oder durch Locks geschützten Abhebung könnte zwischen dem Auslesen des alten Saldowerts und dem Zurückschreiben des neuen Werts ein anderer Thread den Wert verändern; dies hätte einen inkonsistenten Saldo zur Folge.

Die lock-freie Lösung hat jedoch einen entscheidenden Nachteil: Sie erlaubt es nicht, bei Bedarf über mehrere Konten hinweg zu synchronisieren, da wir von Außen keinen Einfluss

auf den von der **AtomicReference** geschützten Bereich haben. Im Falle einer Überweisung wollen wir jedoch sicherstellen, dass während ihrer Ausführung Geldentnahme und Geldgutschrift als *atomare* Operation auf zwei verschiedenen **Konto**-Objekten durchgeführt werden. Wir schwenken daher auf die klassische synchronized Variante um:

```
public class Account...
  private BigDecimal balance a BigDecimal.ZERO;
  public synchronized void deposit(BigDecimal amount) {
    balance = balance.add(amount);
  }
  public synchronized void withdraw(BigDecimal amount) {
    balance = balance.subtract(amount);
  }
  public synchronized BigDecimal getBalance() {
    return balance;
  }
```

Nun kann unser **MoneyTransfer**-Objekt den gleichen Locking-Mechanismus verwenden, um die Überweisung atomar zu machen:

```
public class MoneyTransfer...
  public void execute(Bank bank) {
    Account source = bank.accountByNumber(from);
    Account target = bank.accountByNumber(to);
    synchronized (source) {
      synchronized (target) {
        doTransfer(source, target);
      }
    }
  }
  private void doTransfer(Account src, Account target) {
    if (src.getBalance().compareTo(amount) < 0)
      throw new InsufficientFundsException();
    src.withdraw(amount);
    target.deposit(amount);
  }
```

So offensichtlich, wie die Lösung aussieht, so falsch ist sie. Wir haben mit dem doppelten **synchronized** nämlich eine potenzielle Verklemmung (engl. *Deadlock*) ins Haus gelassen. Treffen beispielsweise zwei Threads aufeinander, wobei der eine von Konto 1 zu Konto 2 überweist und der andere in die entgegengesetzte Richtung, dann kann es passieren, dass der erste Thread den Lock für Konto 1 sichert, während gleichzeitig der zweite Thread den Lock für Konto 2 ergattert. Von nun an werden beide bis in alle Ewigkeit auf den Lock für das jeweils andere Konto warten müssen. Diesem klassischen Lock-Ordering-Problem können wir abhelfen, indem wir bei allen Mehrfachlocks unseres Programms, auch den zukünftigen, für eine eindeutige, deterministische Lock-Reihenfolge sorgen. Etwa so:

```
public class MoneyTransfer...
  public void execute(Bank bank) {
    Account source = bank.accountByNumber(from);
    Account target = bank.accountByNumber(to);
    Object[] locks = new Object[] { source, target };
    Arrays.sort(locks);
    synchronized (locks[0]) {
      synchronized (locks[1]) {
        doTransfer(source, target);
      }
    }
  }

public class Account implements Comparable<Account>...
  public int compareTo(Account other) {
    return new Long(accountNumber).
      compareTo(new Long(other.accountNumber));
  }
```

Das war harte Arbeit. Trotz der leicht verständlichen Fachlogik mussten wir einiges an Kontextwissen bemühen, um Thread-Sicherheit zu erreichen, Verklemmungen zu vermeiden und dennoch die parallele Ausführbarkeit der Fachlogik zu erhalten. Dabei kann es passieren, dass schon die nächste fachlich motivierte Änderung zusätzlichen Synchronisationsaufwand erfordert oder uns gar zum Wechsel der Synchronisationsstrategie zwingt.

Viele Entwickler, die sich näher mit dem Thema beschäftigen, kommen daher zur Einsicht, dass die systemweite, korrekte Verwendung von Locking als Hauptmechanismus für thread-sichere Programmierung ihre persönlichen Fähigkeiten sprengt. Wer noch zweifelt, möge sich an der thread-sicheren Variante der in [Lee06] beschriebenen **Observer**-Klasse versuchen.

Das Problem mit dem Locking

Der Hauptgrund für die Schwierigkeiten beim vorgestellten Programmieransatz ist die mangelnde *Komponierbarkeit* von thread-sicheren Objekten der gezeigten Art: Nur weil ein Objekt – **Account** – für sich alleine betrachtet thread-sicher ist, können wir es nicht einfach in einem anderen Objekt – **MoneyTransfer** – benutzen und dessen Thread-Sicherheit „erben“. Nein, wir müssen über den Synchronisationsmechanismus von **Account** Bescheid wissen, und darüber hinaus das Wissen über unsere Gesamtstrategie, die geordnete Lockreihenfolge, im Rest des Systems verteilen. Brian Goetz [Goe08] nennt das Kind beim Namen: „Locking is not composable“. Damit bricht unsere wunderbare Abstraktion vom gekapselten Objekt, das ich zu anderen komplexeren Objekten zusammenbauen kann, ohne mir über dessen Implementierung Gedanken zu machen, beim Einsatz von Locks zusammen! Was tun?

Zur Hilfe kommen Programmierparadigmen, welche zum Großteil altbekannt sind, jedoch im Zeitalter von immer mehr Rechenkernen ihre Renaissance erleben. Schauen wir einige davon an.

Transactional Memory

Ein denkbarer Ausweg aus dem Dilemma tut sich auf, wenn wir eine zentrale Idee aus der Welt der Datenbanken auf den Hauptspeicher übertragen: der Heap wird zur transaktionalen Datenmenge. Wir erlauben die Deklaration einer atomaren Codesequenz, deren Ausführung ähnliche Eigenschaften haben soll wie eine ACID-Datenbanktransaktion:

- *Atomc:* Entweder tritt jede Zustandsänderung in Kraft oder keine.
- *Consistent:* War der Zustand vor der Ausführung konsistent, so ist er es danach auch.
- *Isolated:* Die Auswirkungen der Codesequenz auf den Programmzustand bekommen andere Threads erst nach erfolgreichem Abschluss zu Gesicht.
- *Durable:* Dauerhaft im Sinne einer Datenbank sind atomare Operationen im Hauptspeicher nicht; dafür sollten sie für alle Threads vollständig sichtbar sein (*safely published*).

Ergänzt man diese Eigenschaften um Schachtelbarkeit, optimistisches Rollback und automatische Wiederholung der atomaren Sequenz im Falle einer Kollision, so erhält man ein intuitives Programmiermodell, das Verklemmungen ausschließt und spürbar einfacher zu verwenden ist als explizite Locks. In Pseudo-Java – man beachte das neue Schlüsselwort **atomic** – sähen Konto und Überweisung folgendermaßen aus:

```
public class Account...
  public void deposit(BigDecimal amount) {
```

```
      atomic {
        balance = balance.add(amount);
      }
    }
    public void withdraw(BigDecimal amount) {
      atomic {
        balance = balance.subtract(amount);
      }
    }

  public class MoneyTransfer...
    public void execute(Bank bank) {
      atomic {
        Account source = bank.accountByNumber(from);
        Account target = bank.accountByNumber(to);
        if (source.getBalance().compareTo(amount) < 0)
          throw new InsufficientFundsException();
        source.withdraw(amount);
        target.deposit(amount);
      }
    }
```

Der entscheidende Unterschied zur Locking-Lösung ist, dass hier das Überweisungsobjekt nichts darüber wissen muss, wie Synchronisation in den verwendeten Konten funktioniert. Wir gewinnen damit die Komponierbarkeit unserer lieb gewonnenen Objekte zurück!

Doch leider bekommen wir diese Bequemlichkeit nicht geschenkt. Mehrere Nachteile transaktionalen Speichers lassen sich aufzählen:

- Der Ansatz bietet keine Hilfestellung bei der Parallelisierung unserer Algorithmen und Programme.
- Innerhalb von atomaren Codeteilen dürfen keinerlei Seiteneffekte (z. B. Schreiben in Datenbank oder Dateisystem, GUI-Ausgabe) auftreten.
- Verklemmungen sind zwar ausgeschlossen, Fortschritt im Programmfluss ohne Fortschritt in der Programmlogik, sogenannte *Livelocks*, ist jedoch immer noch möglich.

Und dann ist da noch die Frage der Praxistauglichkeit. Hardware-basiertes TM (*Transactional Memory*) existiert bislang nur in den Forschungslaboren der Chip-Hersteller. Effiziente Implementierungen von STM (*Software Transactional Memory*) sind schon seit geraumer Zeit Forschungsthema, was die Vermutung nahelegt, dass es sich um ein nicht-triviales Problem handelt. Dennoch gibt es bereits Java-Bibliotheken, die STM zumindest in Grundzügen aus Java heraus ermöglichen [deuce, multiverse]. Der JVM-basierte Lisp-Dialekt *Clojure* [clojure] bietet STM gar als Grundmechanismus, wenn auch mit einem etwas anderen Fokus, nämlich als Ergänzung des Ansatzes, ausschließlich mit nicht-veränderlichem Zustand (*Immutable State*) zu arbeiten.

Immutability

Locking und andere Synchronisationskonzepte benötigen wir, weil wir von unterschiedlichen Threads aus auf gemeinsamen veränderlichen Zustand zugreifen wollen. Warum also nicht einfach auf veränderlichen Zustand verzichten und damit zahlreiche Nebenläufigkeitsprobleme mit einem Schlag aus dem Weg räumen? Diese Philosophie der *Immutability*, die von den meisten funktionalen Sprachen als Standardansatz unterstützt wird, klingt für den OO-gewohnten Entwickler wie ein Taschenspielertrick. Schließlich haben alle nicht-trivialen Programme einen internen Zustand, den man irgendwo zugreifen und verändern können muss, oder etwa nicht?

Der entscheidende Kniff bei Systemen, die (fast) ausschließlich mit unveränderlichen Werten arbeiten, ist die konzeptionelle Unterscheidung zwischen der *Identität* eines Objekts und dem *aktuellen Zustand* des Objekts. Während jede „normale" Operation mit unveränderlichen Objekten arbeitet, existiert für Entitäten – Objekte, deren Zustand sich über die Zeit ändert – ein standardisierter Weg, um ihren Zustand zu aktualisieren. Was sich in der Theorie schwierig anhört, ist in der Praxis leichter verständlich; passen wir unser Bank-Beispiel so an, dass es nur noch an einer Stelle mit veränderlichem Zustand zu tun hat und ansonsten mit unveränderlichen Wert-Objekten arbeitet.

Prinzipiell ließe sich dieser Ansatz auch in Java umsetzen. Ein korrekt implementiertes *Immutable Object* in Java erfordert jedoch nicht nur den **final**-Modifizierer an allen Member-Variablen, sondern zudem noch die zugehörigen Getter-Methoden, den entsprechenden Konstruktor sowie eine ordentliche Implementierung von **equals()** und **hashCode()** [Bloch08]. Wegen dieses hohen Schreibaufwands wechseln wir an dieser Stelle des Artikels zu Javas schreibfaulem Bruder Groovy [groovy]. Eine unveränderliche **Account**-Klasse sieht in Groovy so aus:

```
@Immutable
class Account {
  BigDecimal balance
  long accountNumber
  static Account create(long accountNumber) {
    new Account(balance: 0.0, accountNumber: accountNumber)
  }
  Account deposit(amount) {
    cloneWithChange(balance: balance + amount)
  }
  Account withdraw(amount) {
    cloneWithChange(balance: balance - amount)
  }
  private Account cloneWithChange(changes) {...}
}
```

Die Annotation **@Immutable** sorgt während der Kompilierung dafür, dass all das im Bytecode hinzukommt, was die JVM für ein ordentliches unveränderliches Objekt benötigt. Der große logische Unterschied zur veränderlichen **Account**-Variante ist, dass Methoden, die den Wert – d. h. den Saldo – des Kontos verändern, jetzt ein neues **Konto**-Objekt zurückgeben, das bis auf den Saldo selbst eine Kopie des ursprünglichen Objekts darstellt. *Kopie* ist hierbei lediglich als Konzept zu verstehen; aus Effizienzgründen setzt man häufig auf andere Implementierungen, wie weiter unten erläutert wird.

Um später feststellen zu können, aus welcher Version eines Kontos der veränderte Kontowert hervorgegangen ist, spendieren wir der Klasse einen Versionszähler:

```
class Account...
  long version
  Account incrementVersion() {
    cloneWithChange(version: version + 1)
  }
```

Die **Account**-Klasse ist somit eine reine Werte-Klasse; das Verwalten der Konto-*Identitäten* überlassen wir der Bank:

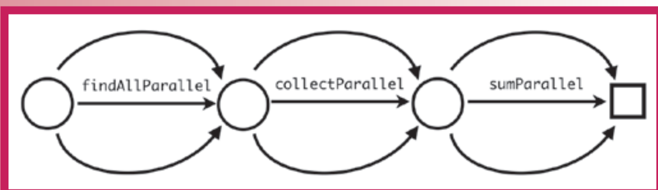
```
class Bank...
  private final accounts = new ConcurrentHashMap()
  Account createNewAccountFor(String customer) {
    Account account = ...
    accounts[account.accountNumber] = account
  }
  Account accountByNumber(long accountNumber) {
    accounts[accountNumber]
  }
  synchronized boolean update(Account... accsToUpdate) {
    if (accsToUpdate.any {acc ->
      acc.version != accounts[acc.accountNumber]?.version
    }) {
```

```
        return false
      }
      accsToUpdate.each {acc ->
        accounts[it.accountNumber] = it.incrementVersion()
      }
      return true
    }
```

Übrig bleibt eine einzige synchronisierte Methode, nämlich die, welche den Update von Konto-Zuständen ermöglicht. Als Parameter nimmt sie eine beliebige Anzahl von Konten, überprüft, ob sich eine der Kontoversionen mittlerweile geändert hat, und wenn nicht, ersetzt sie den Zustand aller übergebenen Konten und erhöht deren Versionszähler. Dieses Vorgehen erlaubt die gleichzeitige und atomare Zustandsänderung mehrerer Konten, wie wir sie in unserer Überweisung benötigen:

```
@Immutable
class MoneyTransfer {
  BigDecimal amount
  long from, to
  boolean execute(Bank bank) {
    while(true) {
      Account source = bank.accountByNumber(from);
      Account target = bank.accountByNumber(to);
      if (source.balance < amount)
        return false
      source = source.withdraw(amount)
      target = target.deposit(amount)
      if (bank.update(source, target))
        return true
    }
  }
}
```

Auch hier erkennt man wieder das Muster optimistischer Operationen: Diese werden so lange wiederholt, bis sie erfolgreich sind, was bei wenigen zu erwartenden Kollisionen fast nie notwendig wird.



Der große Nachteil von Systemen, die auf unveränderliche Werte setzen, ist, neben dem für OO-Entwickler ungewohnten Design, die Performance – zumindest dann, wenn man den Klonvorgang bei verändernden Funktionen naiv als *Deep Copy* implementiert. In diesem Falle benötigt man nicht nur übermäßig viel Speicher, der anschließend vom Garbage Collector wieder freigegeben werden muss, sondern das Kopieren an sich braucht spürbar mehr Prozessorzyklen als eine „normale“ Zustandsänderung. Tatsächlich lässt sich jedoch ein Objektklon sehr ressourcenschonend programmieren, indem man lediglich die Referenz auf den Vorgänger und zusätzlich das Zustandsdelta speichert. Sprachen, die auf Immutability als Hauptparadigma setzen, benutzen daher effizient implementierte unveränderliche Datenstrukturen, z. B. Clojures *Persistent Data Structures*.

Fazit und Ausblick

Javas klassischer Ansatz, die von nebenläufigen Threads gemeinsam genutzten Objekte mittels Locking zu schützen und zu synchronisieren, hat ein fundamentales Problem: Er ist aufgrund fehlender Komponierbarkeit in größeren Programmen kaum fehlerfrei umsetzbar. Bislang haben wir zwei Alternativansätze betrachtet: Transaktionalen Speicher und unveränderliche Wert-Objekte.

Im zweiten Teil des Artikels wird es darum gehen, wie man die Parallelisierung seines Programms erreicht, denn nur wenn das gelingt, kann man auf bessere Performance hoffen. Dazu betrachten wir u. a. Aktoren, Parallel Collections und Data-Flow-Programmierung.

Literatur und Links

[Bloch08] J. Bloch, Effective Java, Addison-Wesley, 2008
[clojure] `http://clojure.org/`
[deuce] `http://www.deucestm.org/`
[Goe06] B. Goetz, Java Concurrency in Practice, Addison-Wesley, 2006
[Goe08] B. Goetz, Concurrency: Past and Present, `http://www.infoq.com/presentations/goetz-concurrency-past-present`
[groovy] `http://groovy.codehaus.org/`
[Hent10] Th. Henties, Urs Gleim, Multicore Programmierung, in: JavaSPEKTRUM, 4/2010??
[Lee06] E. A. Lee, The Problem with Threads, `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf`
[NeTi10] B. Neppert, St. Tilkov, Einführung in Clojure, in: JavaSPEKTRUM, 4/2010??
[multiverse] `http://code.google.com/p/multiverse/`

Johannes Link ist freiberuflicher Coach für Agile Softwareentwicklung. Mit dem Thema Multi-Core-Programmierung für die JVM beschäftigt er sich intensiv, seit er feststellen musste, dass sein neuer Rechner weniger Gigahertz hatte, als sein vorheriger. E-Mail: business@johanneslink.net