Emotional StampedLock

Ein Lock mit Höhen und Tiefen.

Die CoreJava Packages haben in den letzten Versionen immer mal wieder ein paar Werkzeuge für die Bewältigung von Nebenläufigkeits-Problemen bekommen. Ein neuer Vertreter ist der StampedLock in JDK 8. Ein Fall für ein Lächeln oder Tränen? Wir sehen uns die Sache an.

von Dr. Heinz M. Kabutz und Sven Ruppert

Aus dem JavaEE und dem JavaSE Umfeld wissen wir, was es bedeutet sich mit der Problematik der Nebenläufigkeit auseinander zusetzen. Verschiedenste Ansätze wurden und werden immer noch von akademischer und praktischer Seite her eingeführt. Gab es früher nur ein synchronized, kamen später die Semphoren und ähnliches dazu. Was also ist das Neue an dem StampedLock? Wie ist er zu verwenden? Was gibt es für Besonderheiten?

Why Synchronizers?

Zum Einstig nochmals kurz die Fragestellung, warum eigentlich Synchronisationsmittel verwendet werden müssen.

Mit Hilfe der Synchronizers bleiben verteilte mutable Zustände konsistent. Das ist immer dann notwendig, wenn es keine anderen technischen Möglichkeiten gibt den Zustand des Systems in einer zustandslosen oder rein lokalen, nicht verteilten Version zu realisieren. Zu bedenken ist hier, dass immutable Zustände den Garbage-Collector stark beanspruchen können und nicht verteilte/lokale Zustände einen höheren Speicherverbrauch zu Folge haben. Manche unserer Kunden haben HashMaps mit Dateninhalt in der Größenordnung von um die hundert GB.

Coarse Grained Locking

Wie also geht man nun mit der Problematik des Locking um? Eine einfache Methode ist das harte / grobe Locking. Hierbei hat man sicherlich eine Menge Probleme umgegangen, jedoch stellt sich auch nicht der gewünschte Erfolg ein. Es wird mehr oder weniger alles seriell abgearbeitet, was zur Folge hat, dass nur ein Core der verfügbaren CPU´s mit der Lösung der Aufgabe beschäftigt werden kann. (Abb. 1)

001.png

Abb. 1: Auslastung fast nur ein Core

Ziel ist es also eine möglichst ausgeglichene Lockingstrategie, die es ermöglicht möglichst viele Cores gleichzeitig zu verwenden.

„Good“ and „Bad“ Context Switches

Gehen wir also im folgendem davon aus, dass wir eine erhöhte Auslastung der Cores erreichen. Das bedeutet allerdings auch, dass der jeweilige Context für den Thread gewechselt werden muss. Man spricht in diesem Zusammenhang vom ContextSwitch. Nun hat es die Natur der Dinge so an sich, dass es gute und schlechte ContextSwitches gibt. Was ist nun der „Gute“ und was ist der „Schlechte“ ContextSwitch?

Der Gute – ContextSwitch:

Hier sprechen wir von einem Thread, der seine zur Verfügung stehende Rechenzeit voll ausnutzen konnte. Solch ein Thread Context kann normalerweise innerhalb von einem clock cycle ein „swap out“ erfahren und ist offiziell unter dem Begriff „Involuntary Context Switch“ bekannt.

Der Schlechte – ContextSwitch:

Wir sprechen von einem schlechten ContextSwitch wenn der Thread stoppen muss, da eine benötigte Ressource nicht zur Verfügung steht, die von einem anderen suspendierten Thread blockiert wird. Man spricht auch von einem „Voluntary Context Switch“. Solch ein Wechsel kostet von 1000 bis hin zu n*10.000 clock cycles.

Fine Grained Locking

Java gibt einem die Möglichkeit mit dem Schlüsselwort synchronized zu locken. Aus Sicht eines Entwicklers ist das ein sehr einfach anzuwendendes Mittel, dabei ergeben sich aber leider zumeist schlechte ContextSwitches. Was im TaskMonitor (Abb 2.) nach Verschwendung von ca 50% System Time aussieht ist in der Realität jedoch noch meist viel schlimmer. In dem dort verwendeten Beispiel wurde eine Aufgabe anstelle von 28 Sec in 745 Sekunden erledigt. Es hat also 30-mal länger als notwendig gedauert. Was also auf keinen Fall passieren sollte, ist die Anwendung einer zu feinen Lockingstrategie. Das wird sich garantiert schlecht auf die Performance des Gesamtsystems auswirken.

002.png
Abb. 2: 50% System Time

Independent Tasks With No Locking

Das genaue Gegenteil ist die Verwendung von ausschließlich lokalem Speicher pro Thread. Hier gibt es keine Kollisionen, was zu einer 100% Auslastung aller vorhandenen Cores führt. (Abb. 3) Dieses kann als idealer Zustand angenommen werden. Nur leider ist das eher selten zu erreichen in einem größeren System.

003.png
Abb. 3: Independent Tasks

Nonblocking Lock-free Algorithms

Alle Algorithmen die auf Locking basieren sind beeinträchtigt in ihrer Skalierbarkeit. Das wurde durch Amdahl´s und Little´s Laws dargestellt. Es stellt sich nun die Frage, wie man Algorithmen ohne Locking konstruiert. Hier gibt es zwei Ausrichtungen. Die erste ist die Gruppe der „Nonblocking“ Algorithmen. Das sind Algorithmen, bei denen ein Fehler innerhalb des Ablaufs oder das suspendieren eines Threads nicht dazu führen kann, dass andere Threads blockiert werden. Die zweite Gruppe ist die Gruppe der „Lock-Free“ Algorithmen. Hierbei kann bei jedem Schritt ein Thread bei der Durchführung der Aufgabe einen Fortschritt erzielen.

StampedLock

Die Motivation hinter der Klasse StampedLock ist es, die Locks zu trennen nach lesenden und schreibenden Zugriffen. Damit kann man Teile der Anwendung entkoppeln. Es gibt im JDK noch die Klasse ReentrantReadWriteLock. Der Nachteil dieser Implementierung ist die Entscheidung, dass immer pessimistisches Locking verwendet wird. Damit ist ein Teil der möglichen Geschwindigkeitsoptimierung schon per Definition nicht mehr erreichbar. Ein weiterer Nachteil der Implementierung der Klasse ReentrantReadWriteLock ist, das sie zu starvation neigt, wenn gemischt Reader und Writer Threads auf den Lock zugreifen.

Die Nachteile sind zum Teil die Vorteile von StampedLock. Diese Implementierung bietet ein optimistisches Locking für lesende Zugriffe an, das allerdings einfach in pessimistisches Locking überführt werden kann. Schreibende Zugriffe sind immer pessimistisch, arbeiten also mit exclusive

locks und nicht zu vergessen ist, dass ein StampedLock non-reentrant (eintritts- bzw ablaufinvariant) ist.

Kurzer Blick auf das ReadWriteLock Interface

Wie gerade beschrieben sind alle Blocks von ReadWriteLock immer pessimistisch. Hierbei ist zu unterscheiden, dass die Methode writeLock() exklusiv ist. Das bedeutet, dass sich nur jeweils ein Threads zur selben Zeit in dieser Methode befinden kann. Die Methode readLock() kann allerdings von n Threads gleichzeitig durchlaufen werden. Das ist sehr günstig für die Teile, die aus mehrheitlich lesenden Zugriffen bestehen und resultiert in einer besseren Performance. Listing 1 zeigt ein Beispiel für die Implementierung mittels ReadWriteLock.

Listing 1

```
public class BankAccountWithReadWriteLock {
    private final ReadWriteLock lock =
            new ReentrantReadWriteLock();
    private double balance;
    public void deposit(double amount) {
            lock.writeLock().lock();
            try {
                    balance = balance + amount;
            } finally { lock.writeLock().unlock(); }
    }

    public double getBalance() {
            lock.readLock().lock();
            try {
                    return balance;
            } finally { lock.readLock().unlock(); }
    }
}
```

Ende

Wenn man die Anzahl der clock cycles ansieht, die bei der Verwendung dieses Konstruktes auftreten, dann sollte der ReadLock schon etwas mehr Arbeit leisten. Über den Daumen gerechnet sprechen wir von ca 2000 Recheneinheiten. Nur so rechnet sich der Gesamtverbrauch der benötigten Rechenleistung. Mit anderen Worten, dieses macht nur bei länger lesenden Zugriffen Sinn, was in der hier gezeigten Implementierung BankAccount nicht der Fall ist.

ReentrantReadWriteLock Starvation

Wie aber kommt es zu der angemerkten starvation bei der Klasse ReadWriteLock? Bei Java5 gab es von der Priorität her keinen Unterschied zwischen lesenden und schreibenden Zugriffen. Da aber viele Leser zur selben Zeit das ReadLock besitzen konnten, passierte es, dass ein neuer Leser das ReadLock bekam bevor ein zuerst wartendes WriteLock bedient wurde. So mussten die WriteLocks, die ja einen exklusiven Lock benötigten, in manchen Fällen unendlich warten. Das

kann man sich in etwa so vorstellen: Als Autofahrer stehend vor einem Zebrastreifen ohne Ampel an einer Universität wartend. Da kann definitiv länger dauern!

Seit Java6 hat man versucht das zu lösen indem man Leser warten lässt wenn vor ihnen ein Schreiber einen Antrag auf das Lock gestellt hat. Hier ist das Problem nicht so offensichtlich, aber in unseren Experimenten haben wir entdeckt, dass es zu starvation führen kann wenn wir einen Leser haben und viele Schreiber. Der Leser bekommt dann fast nie den Lock.

Also, ReentrantLock, ReentrantReadWriteLock oder synchronized? Was sollte man verwenden? Es ist sehr viel einfacher synchronized zu verwenden, denn es wird automatisch durch die Syntax der Sprache bestimmt wann das Lock losgelassen werden soll. Leider ist die Verwendung des relativ einfachen Idioms von ReentrantLock oftmals falsch umgesetzt. Das äussert sich darin, dass oft das lock/unlock entweder ohne try/finally verwendet wird, oder nicht korrekt eingesetzt wird. (siehe Listing 2)

Listing 2

```
synchronized(this){ /* some operation */ }

//richtige Verwendung
rwlock.writeLock().lock();
try{
   /* some operations */
}finally {
   rwlock.writeLock().unlock();
}

//falsche Verwendung
try{
   rwlock.writeLock().lock();
   /* some operations */
}finally {
   rwlock.writeLock().unlock();
}

//falsche Verwendung
rwlock.writeLock().lock();
/* some operations */
Rwlock.writeLock().unlock();
```

Ende

StampedLock

Nun zum StampedLock (Listing 3). StampedLock ermöglicht uns „Optimistic Locking“, wodurch wir uns den „Bad Context Switch“ bei den Lesenvorgängen sparen können. Dadurch bekommen wir eine viel bessere Performanz bei Lesevorgängen als mit dem älteren ReentrantReadWriteLock. In der ersten Beta Version des StampedLock haben wir ein

Starvationproblem entdeckt. Das wurde zum Glück behoben, sodass wir dies nun für performante Implementierungen einsetzen können. Performanter ist der Einsatz von StampedLock sicherlich, aber die Idiome sind leider noch schwieriger zu verstehen. In Listing 3 ist ein Ausschnitt der zur Verfügung stehenden Methoden. Die Methoden writeLock() und readLock() liefern „long“ Werte als Ergebnis zurück. Dies ist der zum Lock korrespondierende „Stempel“ den wir später benutzen können um die „unlockWrite()“ Methode aufzurufen. Die pessimistischen read und write Methoden sind sich sehr ähnlich.

Listing 3

```
public class StampedLock { //partial

   // pessimistic exclusive locks
   long writeLock()
   long writeLockInterruptibly() throws InterruptedException

   long tryWriteLock()
   long tryWriteLock(long time, TimeUnit unit) throws InterruptedException

   void unlockWrite(long stamp)
   boolean tryUnlockWrite()
   Lock asWriteLock()
   long tryConvertToWriteLock(long stamp)

   // pessimistic non-exclusive
   long readLock()
   long readLockInterruptibly() throws InterruptedException

   long tryReadLock()
   long tryReadLock(long time, TimeUnit unit) throws InterruptedException

   void unlockRead(long stamp)
   boolean tryUnlockRead()
   Lock asReadLock()
   long tryConvertToReadLock(long stamp)
```

Ende

Als Vergleich hier nun der BankAccount von Listing 1 als StampedLock-Version. Unter Verwendung der pessimistischen ReadLocks sieht der Quelltext fast identisch zu Listing 1 aus.

Listing 4

```
public class BankAccountWithStampedLock {
   private final StampedLock lock = new StampedLock();
   private double balance;
   public void deposit(double amount) {
```

```
            long stamp = lock.writeLock();
            try {
                    balance = balance + amount;
            } finally { lock.unlockWrite(stamp); }
    }

    public double getBalance() {
            long stamp = lock.readLock();
            try {
                    return balance;
            } finally { lock.unlockRead(stamp); }
    }
}
```

Ende

Aber kann dieser Ausdruck nicht unter der Verwendung von synchronized/volatile einfacher geschrieben verwenden? In Listing 5 sehen wir, wie man das in unserem Fall machen könnte. Wir brauchen „`volatile`“ in Verbindung mit dem „`balance`“ Feld aus zwei Gründen:

1. Ein Thread könnte einen alten Wert von balance sehen.

2. Auf einer 32-bit Maschine könnte das 64-bit „balance“ Feld in zwei Instruktionen geschrieben werden. Das hat zur Folge, dass in dieser Zeit „unmögliche“ Werte ausgelesen werden können.

Auf der WJAX 2013 erzählte ein Bankenprogrammierer in einem persönlichen Gespräch, dass bei ihrem System aus Versehen €200 Millionen auf ein falsches Konto gutgeschrieben worden sind. Der Kontoinhaber war hoch erfreut und hat sofort €10 Millionen davon auf ein anderes Konto überwiesen (Wer hätte das nicht gemacht)! Die Bank hat diesen Irrtum jedoch schnell bemerkt und die Überweisung von €200 Millionen rückgängig gemacht. Ergebnis war, dass der Kontoinhaber nun ein Minus von €10 Millionen hatte. Fällig wurden dafür Zinsen in Höhe von €12000, zahlbar sofort. Solche Fehler mit teils weitreichenden Folgen, können einem leicht passieren wenn man bei dem Thema Synchronisation nicht aufpasst! Das Schlüsselwort volatile ist in Listing 5 demnach sehr wichtig. Diese Lösung ist nur möglich in diesem Falle, da keine Invarianten zwischen beteiligten Feldern bestehen.

Listing 5

```
public class BankAccountWithVolatile {
    private volatile double balance;
    public synchronized void deposit(double amount) {
            balance = balance + amount;
    }
    public double getBalance() {
            return balance;
    }
}
```

Ende

Das allerdings ist nicht der Normalfall. Leider ist es eher genau anders herum. Es bestehen Beziehungen zwischen Feldern. Diese müssen in mehreren Schritten geändert werden, ergeben aber nur einen Sinn wenn alle Änderungen in Summe atomar ablaufen. Das bedeutet, entweder alle Änderungen finden statt, oder keine. Es handelt sich also um eine transaktionale Betrachtung.

Gehen wir im folgenden Beispiel von einer Koordinate (`MyPoint`, Listing 6) aus, die aus einer x- und einer y-Komponente besteht. Es sollen immer nur gültige Koordinaten dem restlichen System zur Verfügung stehen. Aus diesem Grund ist die Methode move() per exclusive lock geschützt. Man sieht hier, das der Lock eine Marke vom Typ long zurückgibt. Diese Marke ist notwendig um explizit den korrespondierenden Lock wieder zu lösen.

Listing 6

```
public class MyPoint {
   private double x, y;
   private final StampedLock sl = new StampedLock();
   // method is modifying x and y, needs exclusive lock
   public void move(double deltaX, double deltaY) {
           long stamp = sl.writeLock();
           try {
                   x += deltaX;
                   y += deltaY;
           } finally {
                   sl.unlockWrite(stamp);
           }
   }
   //…
```

Ende

Die Methode `optimisticRead()` in Listing 7 hingegen versucht zuerst ein optimistisches Lesen, indem ein optimistic Read angefordert wird (`tryOptimisticRead())` um dann nachfolgend die benötigten Variablen in lokale Variablen (`double currentX = x, currentY =y;`) zu kopieren. Nach dem Kopiervorgang wird überprüft, ob es in der Zwischenzeit einen Schreibzugriff gegeben hat (`sl.validate(stamp)`). Dabei ist es immer zu empfehlen, so wenige Anweisungen wie möglich innerhalb des optimistic Read durchzuführen. Dadurch wird die Wahrscheinlichkeit erhöht, dass es erfolgreich verlaufen wird. Wenn es einen Schreibzugriff gegeben hat, wird ein pessimistic Read angefordert (`sl.readLock()`) und die Variablen werden erneut in lokalen Variablen gespeichert. Nun kann man davon ausgehen, dass die lokalen Variablen einen in sich geschlossenen Zustand haben. Die nachfolgende Verarbeitung kann beginnen (`Math.hypot(currentX, currentY)`).

Dieses Verhalten ist selbst dann noch sinnvoll, wenn eine hohe Anzahl Writer involviert sind. Die Performance ist dann immer noch höher im Vergleich zu der Verwendung von ReentranReadWriteLocks. Sind die optimistic Reads meist erfolgreich lohnt sich der Mehraufwand eines hin und wieder auftretenden pessimistic Read incl der zusätzlichen Kopieraktionen in Summe dennoch. In Summe ist eine Reduktion der gesamt notwendigen clock cycles erfolgt.

Listing 7

```
public class MyPoint {
    //…
    public double distanceFromOrigin() {
            long stamp = sl.tryOptimisticRead();
            double currentX = x, currentY = y;
            if (!sl.validate(stamp)) {
                    stamp = sl.readLock();
                    try {
                            currentX = x;
                            currentY = y;
                    } finally {
                            sl.unlockRead(stamp);
                    }
            }
            return Math.hypot(currentX, currentY);
    }
    //…
```

Ende

Ein weiterer Fall ist der „conditional change“ in Listing 8. Als Ausgangspunkt haben wir die benötigten Variablen in der alten und evtl. neuen Version. Diese werden der Methode `changeStateIfEquals` übergeben. Ziel der Methode ist es, die lokalen Variablen x und y in der Klasseninstanz auf die neuen Werte zu setzen wenn sie einen bestimmten Wert haben. Wir beginnen mit einem pessimistic Readlock und speichern den Marker in der Variablen `stamp` zwischen. Sollte der derzeitige Zustand nicht dem erwarteten entsprechen, beenden wir die Methode und geben per `unlock(stamp)` den vorliegenden Lock frei. Wichtig an dieser Stelle zu wissen ist, dass ein `unlock` Read- als auch Writelocks freigibt. In der Klasse MyPoint ist dieser Zustand erreicht, wenn die old-Values nicht den gewünschten entsprechen. Sollten die old-Values den aktuellen Werten entsprechen, wird versucht den Readlock in ein Writelock zu konvertieren. Der Marker wird in der Variablen writeStamp zwischengespeichert. Sollte der WriteStamp den Werteinhalt ungleich 0 haben, konnte der Readlock in einen Writelock konvertiert werden. Der neue Marker ist nun aktiv, den alten können wir vergessen da er nicht mehr von Bestand ist. Es wird deswegen die Variable `stamp` mit dem Wert des WriteMarkers überschrieben. Es ist nun der richtige Zeitpunkt die neuen Werte zu übernehmen. In unserem Beispiel `x= newX und y= newY`. Die Methode kann nun mit dem Result true terminieren, da ein erfolgreicher Schreibzugriff erfolgte. Die Variablen x und y sind nun in einem weiteren gültigen Zustand. Sollte der WriteStamp allerdings den Werteinhalt 0 haben, konnte der Readlock nicht in einen Writelock konvertiert werden. Das hat zur Folge, dass der bestehende Readlock beendet wird (`unlockRead(stamp)`) und ein neuer Writelock angefordert wird. Ist die Bedingung der while –Schleife immer noch true, wird eine weitere Iteration vorgenommen. Die Bedingung kann false sein, wenn in der Zwischenzeit ein anderer Thread den Zustand geändert hat. Es erfolgt dann ein `unlock(stamp)` und als Returnwert wird ein false geliefert. Gehen wir also nun davon aus, dass der Zustand nicht von einem anderen Thread geändert worden ist, die Bedingung der while-Schleife also noch true ist. Der Versuch in einen Writelock zu konvertieren

wird erfolgreich sein, da es sich schon um einen solchen handelt. Die Variablen werden nun geändert, die Methode liefert true als Ergebnis zurück. Wie auch immer die Methode verlassen wird, der `finally` Block wird immer dafür sorgen, dass der bestehende Lock freigegeben wird.

Listing 8

```
public class MyPoint {
   public boolean moveIfAt(double oldX, double oldY,
                                double newX, double newY) {
           long stamp = sl.readLock();
           try {
                   while (x == oldX && y == oldY) {
                           long writeStamp = sl.tryConvertToWriteLock(stamp);
                           if (writeStamp != 0L) {
                                   stamp = writeStamp;
                                   x = newX; y = newY;
                                   return true;
                           } else {
                                   sl.unlockRead(stamp);
                                   stamp = sl.writeLock();
                           }
                   }
                   return false;
           } finally {
                   sl.unlock(stamp);
           }
   }


}
```

Ende

Fazit:

Die Verwendung von StampedLock ist fast immer vorteilhaft. Tests haben gezeigt, dass der Schreibzugriff sich ähnlich schnell verhält wie bei einem ReadWriteLock. Zu wesentlich mehr Perfomancegewinnen kommt es bei der Verwendung des ReadLocks. Hier werden teilweise beachtliche SpeedUps von 10 bis 100 erreicht. Grund dessen ist die Möglichkeit das optimistic Read verwenden zu können. Wird das allerdings in der Verwendung nicht beachtet, fallen die Leistungswerte stark ab.

Dr Heinz Kabutz schreibt den Java Specialists Newsletter, der heutzutage in 134 Ländern der Welt gelesen wird, was selbst MacDonalds nicht schafft. Seine besondere Interessen sind: Fortgeschrittenes Java, Concurrency, Performanz und Kochen. [1]

Sven Ruppert arbeitet bei der SiteOS AG in München, spricht seit 1996 Java und arbeitet seitdem in nationalen und internationalen Projekten in den Bereichen verteilter Systeme. [2]

Links & Literatur

[1] http://www.javaspecialists.eu
[2] https://www.xing.com/profiles/Sven_Ruppert2