Memory Models

Frederik Zipp

23. Juli 2010

Zusammenfassung

Parallele Programmierung gewinnt zunehmend an Bedeutung. Transformationen des Programmcodes, die von Compiler oder Hardware zu Optimierungszwecken vorgenommen werden, können u. a. zu Wettlaufsituationen bei Speicherzugriffen führen, welche ihrerseits schwer zu findende Programmfehler verursachen können. Anhand einfacher Beispiele werden die Probleme verdeutlicht.

Sogenannte Memory Models legen daher die Regeln fest, ob und in welchem Ausmaß Transformationen zulässig sind. Restriktive Memory Models, als deren Vertreter Sequential Consistency vorgestellt wird, sind für den Programmierer leicht verständlich, unterbinden aber zugleich nahezu jegliche Codeoptimierung, sodass die Leistungsfähigkeit moderner Systeme nur zum Teil ausgeschöpt wird. Als Vertreter der weniger restriktiven Memory Models wird das Happens-Before-Model besprochen. Es lässt Compiler und Hardware deutlich größeren Optimierungsspielraum. Seine Regeln zur Unterbindung unerwünschter Transformationen werden detailliert dargestellt. Abschließend wird das Java Memory Model als Prototyp eines guten Kompromisses zwischen Systemeffizienz und Verständlichkeit in seinen Grundzügen kursorisch abgehandelt.

Inhaltsverzeichnis

1 Vorbemerkung

Der folgende Text ist die schriftliche Ausarbeitung zum Vortrag über das Thema Memory Models im Rahmen des Seminars *Sprachen für Parallelprogrammierung* im Sommersemester 2010. Meine Aussagen stützen sich im Wesentlichen auf zwei Fachartikel zum Thema ([Boe05], [MPA05]). Sofern ich weitere Quellen benutzt habe, sind diese im Text angegeben.

2 Hintergründe

Parallele Programmierung gewinnt in der heutigen Softwareentwicklung zunehmend an Bedeutung. Paralleles Programmieren umfasst zum einen Methoden, um ein Computerprogramm in einzelne Teilstücke aufzuteilen, die nebenläufig ausgeführt werden können, zum anderen Methoden, nebenläufige Programmabschnitte zu synchronisieren. Üblicherweise lässt der Programmierer dabei einzelne Programmteile in voneinander getrennten Prozessen oder Threads ausführen. Hierbei ist es für den Programmierer nebensächlich, ob die einzelnen Programmteile wirklich gleichzeitig von unabhängigen Prozessoren bearbeitet werden, oder ob sie nur quasi-parallel ausgeführt werden, wie es z.B. beim Time-Sharing oder Multi-Tasking der Fall ist.

Es gibt eine Vielzahl von Gründen, parallele Programmierung anzuwenden.

- Effizienzsteigerung: Die Leistungskapazität von zunehmend häufiger eingesetzten Multiprozessorsystemen kann nur auf diese Weise ausgeschöpft werden. Auch in Einprozessorsystemen gibt es Steigerungsmöglichkeit der Leistung, indem z.B. Wartezeiten eines Threads ausgenutzt werden, um einen anderen Thread zu bearbeiten.

- Programmerfordernisse: Ein Server muss mehrere gleichzeitig eintreffende Client-Anfragen bedienen, oder ein einzelner Nutzer nimmt mehrere miteinander verknüpfte Dienste in Anspruch.

Ein wesentliches Merkmal dieser Programmiertechnik ist, dass die verschiedenen Threads sich üblicherweise einen gemeinsamen Speicherbereich teilen. Diese Eigenschaft kann jedoch Ursache für Probleme sein, die bei der herkömmlichen Programmiertechnik unbekannt waren. Typische Konflikte sind im Folgenden aufgeführt:

- Verklemmung (*deadlock*), d.h. zwei Abläufe warten aufeinander und blockieren sich gegenseitig.

- Wettlaufsituation (*race condition*), die im Folgenden näher beschrieben wird.

Eine Wettlaufsituation kann entstehen, wenn zwei Threads zeitgleich über nichtatomare Speicheroperationen auf den selben Speicherplatz zugreifen und eine der beiden Operationen einen Schreibzugriff beinhaltet. Das Ergebnis dieser Wettlaufsituation kann unvorhersehbar sein. Dies wird am Beispiel von Abbildung 1 erläutert.

Thread 1	Thread 2
`a = x`	`b = x`
`a = a + 1`	`b = b + 1`
`x = a`	`x = b`

Abbildung 1: Eine Wettlaufsituation

Der Wert von x beträgt anfangs 1. Wenn die beiden Threads sequentiell ablaufen, dann ist das Endergebnis von x in jedem Fall 3. Bei paralleler Ausführung

hingegen kann es passieren, dass das Endergebnis 2 beträgt, wenn beispielsweise Thread 2 den Wert x liest, bevor Thread 1 den erhöhten Wert zurückgeschrieben hat.

Diese Wettlaufsituationen sind besonders problematisch, da sie nur sporadisch auftreten und schwer zu entdecken sind. Die übliche Maßnahme, um Wettlaufsituationen zu vermeiden, ist die sogenannte Synchronisation. Hierdurch wird sichergestellt, dass die beiden Threads einen bestimmten Programmabschnitt nicht gleichzeitig ausführen. Falls nämlich der eine Thread diesen Programmabschnitt ausführt, muss der andere Thread solange zurücktreten, bis der erste Prozess abgeschlossen ist. Von den vielen eingeführten Synchronisationsmechanismen seien beispielsweise *Locks* und *Mutexes* erwähnt, wie sie beispielsweise von der Threading-Bibliothek *pthreads* angeboten werden.

Diese Synchronisationsmechanismen alleine bieten jedoch keinen vollständigen Schutz gegen das Auftreten von Wettlaufbedingungen, weil der Programmier nicht unter allen Umständen erkennen kann, wann er diese Mechanismen einsetzen muss. In Betracht zu ziehen sind nämlich auch Optimierungsprozesse, die sowohl Compiler als auch Hardware durchführen können. Auf die daraus entstehenden Probleme wird in [Boe05] hingewiesen.

Anfangszustand: x = y = 0

Thread 1	Thread 2
`if (x == 1)`	`if (y == 1)`
`  y = y + 1`	`  x = x + 1`

Abbildung 2: Liegt eine Wettlaufbedingung vor?

Ein entsprechendes Beispiel ist in Abbildung 2 dargestellt. Auf den ersten Blick ist in diesem Beispiel keine Wettlaufsituation erkennbar. Die beiden Variablen x und y können scheinbar keinen anderen Wert als 0 annehmen. Wenn nun aber ein Compiler, im Bestreben den Programmcode zu optimieren, bestimmte Transformationen durchführt, kann der formal gleichwertige Code in Abbildung 3 entstehen.

Anfangszustand: x = y = 0

Thread 1	Thread 2
`y = y + 1`	`x = x + 1`
`if (x != 1)`	`if (y != 1)`
`  y = y - 1`	`  x = x - 1`

Abbildung 3: Nach der Transformation durch den Compiler

Dieser Code lässt leicht erkennen, dass nun eine Wettlaufsituation entstanden ist und als potentielles Endergebnis $x = y = 1$ möglich ist.

Im Interesse der Optimierung nimmt ein Compiler auch häufig ein *Reordering* des Programmcodes vor. Diese Art von Transformation ist zulässig, solange Abhängigkeiten innerhalb eines Threads nicht verletzt werden. Für dem gesamten Programmcode kann dies jedoch das Auftreten einer Wettbewerbssituation bedeuten. Beispielsweise kann der Programmcode in Abbildung 4 vor dem Reordering nie als Ergebnis $r1 = r2 = 0$ haben. Nach dem Reordering durch

den Compiler, in Abbildung 5 dargestellt, ist dies als Ergebnis jedoch durchaus möglich.

Anfangszustand: x = y = 0

Thread 1	Thread 2
`x = 1`	`y = 1`
`r1 = y`	`r2 = x`

Abbildung 4: Ist $r1 = r2 = 0$ ein mögliches Ergebnis nach Ende der Threads?

Anfangszustand: x = y = 0

Thread 1	Thread 2
`r1 = y`	`r2 = x`
`x = 1`	`y = 1`

Abbildung 5: Nach dem Reordering. $r1 = r2 = 0$ ist nun ein mögliches Ergebnis

Nach dem bisher Gesagten wird erkennbar, dass ein Programmierer nicht die volle Kontrolle über die Transformationen hat, die Compiler und Hardware mit seinem Programm vornehmen, und dass sich hierdurch schwer erkennbare Fehler in seinen Programmcode einschleichen können. Dieses Dilemma wird im Folgenden mit Hilfe von Memory Models aufgelöst.

3 Memory Models

Bei einem *Memory Model* geht es ganz allgemein um Regeln für die Zugriffe auf den gemeinsamen Speicher. Wenn sich der Programmierer an die Regeln hält, dann gibt ihm das System Garantien für die Effekte von Speicherzugriffen, damit der Programmierer weiß, was zur Laufzeit geschehen wird, und er die Effekte seiner Speicheroperationen vorhersehen kann [KrLa08].

Im Folgenden werden drei ausgewählte Memory Models vorgestellt:

- Sequential Consistency
- Happens-Before Memory Model
- Java Memory Model

3.1 Sequential Consistency

Unter den Memory Models ist Sequential Consistency das einfachste Modell. Es kommt in seiner Betrachtungsweise dem menschlichen Vorstellungsvermögen am weitesten entgegen. Dem Programmierer gegenüber stellt es sich so dar, als ob die einzelnen Threads wie in einem Einprozessorsystem Zeitscheiben zugeteilt bekommen. Nachdem ein Thread einige Operationen ausgeführt hat wird er von einem anderen Thread abgelöst, der wiederum einige Operationen ausführt.

Alle Schreiboperationen der zuvor ausgeführten Threads sind für die zeitlich nachfolgenden Threads sichtbar.

Aufgrund dieser strikten Regeln gehört Sequential Consistency zu den restriktivsten Memory Models. Compiler und Hardware finden in diesem Modell nur geringen Spielraum für eine Optimierung des Programmcodes. Zum Beispiel ist es nicht erlaubt, innerhalb eines Threads irgendein beliebiges Paar von Anweisungen in der Reihenfolge zu vertauschen. Dies trifft sogar zu, wenn keinerlei Abhängigkeit zwischen den Anweisungen besteht. Die Einfachheit dieses Modells wird mit Verlust an Effizienz bezahlt.

3.2 Happens-Before Memory Model

Das Happens-Before Memory Model fällt in die Gruppe der weniger restriktiven Memory Models.

Die zentralen Begriffe des Happens-Before Memory Models sind in Abbildung 6 dargestellt und werden wie folgt definiert:

- Programmordnung: Abfolge der Aktionen, definiert durch die Kanten innerhalb der Threads.
- Synchronisationsaktionen: umfassen Aktionen wie Locks, Unlocks sowie Lese- und Schreibzugriffe auf volatile Variablen.
- Synchronisationsordnung: die Totalordnung aller Synchronisationsaktionen innerhalb eines Programmablaufes.
- Synchronizes-With-Ordnung: Eine Unlock-Aktion auf zum Beispiel einem Monitor m *synchronizes-with* allen nachfolgenden Lock-Aktionen eines jeden Threads auf m. Dabei ist „nachfolgend“ durch die Synchronisationsordnung definiert. Eine solche Beziehung wird in der Abbildung exemplarisch durch eine Kante *sw* dargestellt.
- Happens-Before-Ordnung: *happens-before* ist die transitive Hülle der Programmordnung und der *synchronizes-with*-Ordnung. Eine Beziehung wird in Abbildung 6 durch die Kanten *hb* dargestellt. Jede *sw*-Kante ist auch eine *hb*-Kante.

Unter Zugrundelegung dieser Definitionen ist ein Programmablauf dann zulässig, wenn folgende Bedingungen erfüllt sind:

- Intra-Thread-Konsistenz: Für jeden Thread sind die Aktionen, die dieser Thread in der Ausführung durchführt, identisch mit denjenigen, die dieser Thread als isolierter Thread in der Programmordnung erzeugen würde.
- Happens-Before-Konsistenz: eine Leseaktion darf keine Schreibaktion wahrnehmen, die in der Happens-Before-Ordnung nach dieser Leseaktion folgt. Des Weiteren darf eine Leseaktion keine Schreibaktion wahrnehmen, bei der sich in der Happens-Before-Ordnung zwischen dieser Schreibaktion und der Leseaktion eine weitere Schreibaktion befindet.
- Synchronisationsordnungs-Konsistenz: Die Synchronisationsordnung muss übereinstimmend mit der Programmordnung sein. Des Weiteren muss jede Leseaktion einer volatilen Variable die letzte Schreibaktion auf diese Variable sehen, die vor ihr in der Synchronisationsordnung war.

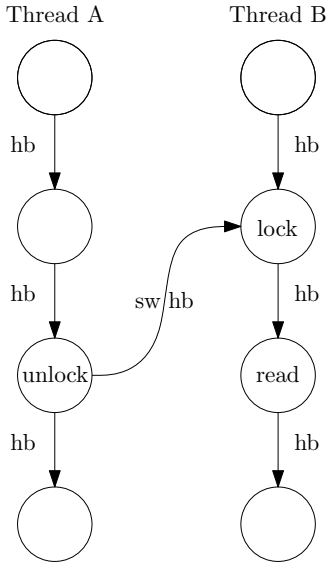

Abbildung 6: Ein Happens-Before-Graph. Die Knoten sind Aktionen (Lese- und Schreibzugriffe oder Synchronisationsaktionen wie *lock*, *unlock*), eine *synchronizes-with*-Beziehung (*sw*) führt eine zusätzliche *happens-before*-Kante (*hb*) ein.

Das Happens-Before-Model gibt Compiler und Hardware deutlich mehr Optimierungsfreiräume. Gleichzeitig wird aber auch erkennbar, dass die komplexen Regeln die Anwendung für den Programmierer unübersichtlicher machen.

3.3 Java Memory Model

Das hier besprochene Java Memory Model entstand im Zuge der Entwicklung von Java 5.0. Eine ausführliche Beschreibung der Anforderungen an das neue Modell findet man bei [MaPu04]. Ziel der Entwickler war, einen ausgewogenen Kompromiss zwischen der Handhabbarkeit für Programmierer einerseits und hoher Implementierungsflexibilität für Systementwickler andererseits zu finden.

Das Java Memory Model gehört in die Gruppe der weniger strikten Memory Models. Es basiert auf dem bereits vorgestellten Happens-Before-Model, wobei bekannte Schwächen dieses Modells vermieden werden.

Neu und einzigartig ist ein iterativer Ansatz, um erlaubte Ausführungen zu erstellen. Die Sicherheits- und Zuverlässigkeitsphilosophie von Java wurde dergestalt umgesetzt, dass eine Klasse von Ausführungen, die in einer Wettlaufsituation münden, identifiziert und als unzulässig deklariert wird.

Zugleich wurde großer Wert darauf gelegt, dass die üblichen von Compiler oder Hardware durchgeführte Transformationen weiterhin zulässig bleiben. Diese sind jedoch für den Programmierer transparent und er braucht sich um diese Vorgänge nicht zu kümmern, solange er in seinem Programmcode auf eine korrekte Synchronisation achtet und Wettlaufsituationen vermeidet.

Außerdem wurde im Interesse der Programmierer eine allzu große Verkomplizierung des Modells vermieden.

Schließlich treffen nach Ansicht der Entwickler dieses Modells viele der bearbeiteten Probleme auch auf andere Programmiersprachen mit Multithreading-untestützung zu.

4 Fazit

Memory Models sind eine unentbehrliche Voraussetzung für jedes System, das Multithreading mit Zugriff auf gemeinsam genutzte Speicherplätze unterstützt. Memory Models sollen verhindern, dass von Compiler oder Hardware unzulässige Transformationen des Programmcodes vorgenommen werden, die in der Folge zu einer schwierig zu erkennbaren Fehlerquelle werden können. Memory Models bewegen sich zwischen den Polen Effizienz des Systems einerseits und Verständlichkeit für den Programmierer andererseits. Restriktive Memory Models, als deren Vetreter Sequential Consistency vorgestellt wurde, lassen die Kapazität moderner CPUs zum großen Teil brach liegen, sind aber einfach zu handhaben. Als Prototyp für einen vernünftigen Kompromiss zwischen den beiden widerstreitenden Anforderungen kann das Java Memory Model angesehen werden.

Literatur

[Boe05] Hans J. Boehm. Threads cannot be implemented as a library. *In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, volume 40, pages 261–268, New York, NY, USA, June 2005. ACM.

[KrLa08] Klaus Kreft, Angelika Langer. Überblick über das Java Memory Model, *Java Magazin*, August 2008 http://www.AngelikaLanger.com/Articles/EffectiveJava/38.JMM-Overview/38.JMM-Overview.html

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. *In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[MaPu04] Jeremy Manson and William Pugh. Semantics of Multithreaded Java. *Technical Report CS-TR-42115*, Dept. of Computer Science, University of Maryland, College Park, March 2004

[JLS3] Sun Microsystems, Inc. The Java Language Specification, Third Edition. Chapter 17, *Threads and Locks* http://java.sun.com/docs/books/jls/third_edition/html/memory.html