

The Java Virtual Machine und The Java Memory Model



**VORTAG VON GEORG EUTERMOSER
UND SARAH BREINING**

AM 11.JULI.2008

Im Rahmen des Proseminars "Nebenläufige Programmierung"

Gliederung



0 Java Virtual Machine (JVM)

0.1 Was ist die JVM?

0.2 Was macht die JVM?

0.3 Sicherheitsvorteile der JVM

0.4 Optimierungen

1 The Java Memory Model (JMM)

1.1 Was ist ein Java Memory Model und wozu braucht man es?

1.2 Happens-before

1.3 Publication

1.4 Initialization safety

Java Virtual Machine



0.1 Was ist die JVM?



- Teil der Java-Laufzeitumgebung (JRE)
- Die Bestandteile der JVM sind:
 - Klassenlader
 - Speicherverwaltung und automatische Speicherbereinigung (garbage collection)
 - Ausführungseinheit

0.1.1 Der Klassenlader



- entscheidet, ob eine Klasse geladen werden darf
- stellt sicher, dass Anwendungen keine Systemklassen (speziell den Klassenlader selbst) überschreiben

- Die drei wichtigsten Lader:
 - System-Klassenlader
 - Erweiterungs-Klassenlader
 - Applikations-Klassenlader

0.2 Was macht die JVM?



- führt den Java-Bytecode aus
- dient als Schnittstelle zur Maschine und zum Betriebssystem
- koordiniert zwischen JMM und dem plattformspezifischen Memory Model

0.3 Sicherheitsvorteile der JVM



- **Bytecode Verifier**
 - überwacht das Programm zur Laufzeit
 - Referenzen zeigen immer auf gültige Speicherplätze
 - überwacht Dateninitialisierung und Typensicherheit
 - verhindert Lesen und Schreiben außerhalb von Arraygrenzen

→ kein Buffer-Overflow, der von Hackern genutzt wird um ein System unter ihre Kontrolle zu bringen!!

0.3.1 Sandbox



- Die Sandbox ist ein Sicherheitsmodell bei der Ausführung von Applets im Internet
- Sie verhindert, dass ein Applet Zugriff auf lokale Ressourcen erhält
- Sie besteht aus:
 - Klassen Lader
 - Byte Code Verifier
 - Security Manager

0.4 Optimierung



- **Just in Time Compiler (JITC):** Im Gegensatz zu Ahead of Time Compilern übersetzt die JVM den Code erst zur Laufzeit des Programms und nicht schon vor der Ausführung.
- **Vorteile:**
 - konstante Werte können als solche erkannt und behandelt werden
 - keine falsche Sprungvorhersage
- **Nachteile:**
 - Bei jedem neuen Start wird der Bytecode aufs neue kompiliert
 - Bei einmal aufgerufenen Methoden uneffektiv

0.4.1 Sprungvorhersage



- **Das Problem:**
- Bei einem Sprung werden schon geladene Daten im Prozessor überflüssig
- **Lösung:**
- Vorhersage, ob ein bedingter Sprung ausgeführt wird
- Zieladresse eines Sprunges ermitteln

- **Programm**

boolean entscheid;

...

entscheid= ...;

...

if(entscheid) Möglichkeit1

else Möglichkeit2

0.4.2 Hotspot Optimierung



- Weiterentwicklung von JiTC
- der Bytecode wird zur Laufzeit nur interpretiert
- Häufig aufgerufene Codeabschnitte (Hotspots) werden dann JiT compiliert
- Vorteile:
 - Bessere Optimierung möglich, da nur unter den gerade geladene Bedingungen der Code optimiert wird. Wenn Code nachgeladen wird muss evtl. neu compiliert werden.
 - Mehraufwand während der Laufzeit durch JiT Compilierung vernachlässigbar, da nur wenige Stellen compiliert werden

0.4.2 Hotspot Optimierung



- **Inlining**
 - Kurze Methoden werden in den Rumpf des Aufrufers eingefügt, anstatt angesprungen zu werden

- **Loop-Unrolling**
 - Bei kurzen Schleifen werden die einzelnen Schleifendurchläufe sequenziell ohne Rücksprung abgearbeitet

0.4.2 Hotspot Optimierung



- **Dead Code Elimination**
 - Ungenutzte Anweisungen werden auf Bytecodeebene entdeckt und verworfen

- **Peephole-Optimierung**
 - Hierbei werden z. B. redundante Speicherzugriffe eliminiert und Registerzugriffe optimiert

0.5 Abschottung von Threads



- JVM bietet die Möglichkeit der User Thread Funktionalität (green Threads)
- Vorteile:
 - Keine Betriebssystemunterstützung notwendig
 - Threadwechsel sehr effizient
- Nachteil:
 - einzelne fehlerhafte Prozesse der JVM können vom Betriebssystem nicht beendet werden

Java Memory Model



1.1 Was ist ein Memory Model und wozu braucht man es?



- Es bestimmt die Sichtbarkeit der Variablen für die Threads eines Programmes:
 - Variablen können in Caches lagern und unsichtbar sein
 - Befehle können in anderer Reihenfolge ausgeführt werden
 - Variablen die in einem Cache gespeichert sind können unsichtbar für andere Prozessoren sein
- Bei sequenziellen Programmen übernimmt das das JMM automatisch
- Jedoch: bei Multithreading nicht!

1.1.1 Platform memory models



- Prozessoren haben ihre eigenen Caches
 - Nicht sichtbar für andere Prozessoren
 - Speicherbarrieren koordinieren Datasharing zwischen den Kernen
 - PMM <> JMM
 - In Java: JVM koordiniert zwischen JMM und dem plattformspezifischen Speicher-Modell
- keine Barrieren sondern Synchronisation zur Koordination

1.1.2 Reordering



- **Sequenzielle Beschaffenheit:**

Modell eines Programms, das einen festen Ablauf der Befehle annimmt, bei dem jeder Befehl alles vorher geschriebene sieht

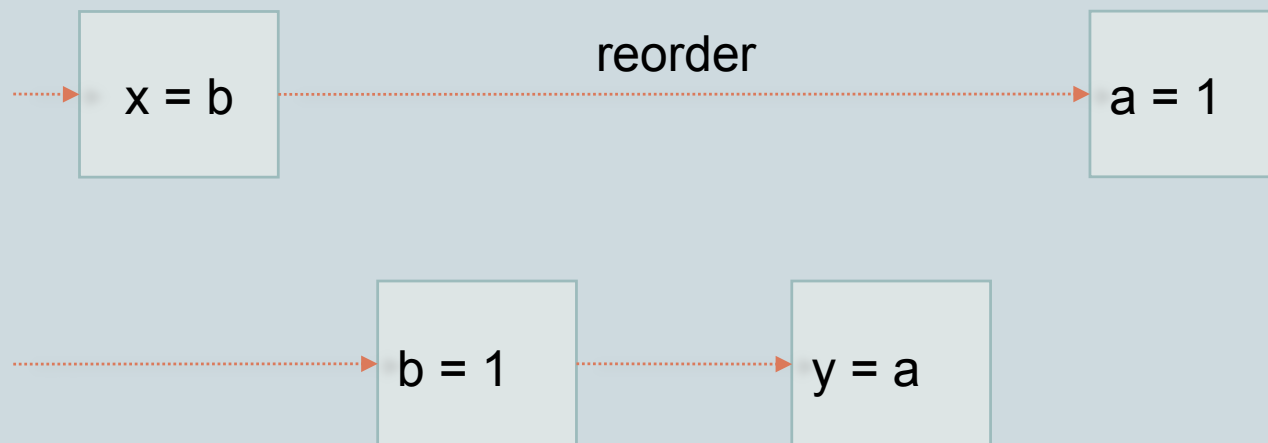
- **Wirklichkeit:**

Java ordnet die Befehle zur Optimierung nach Belieben neu

1.1.2 Reordering



- Instruktionen werden umgeordnet, solange sich die Semantik des Programms nicht ändert
- Cache speichert Variablen in anderer Reihenfolge zurück
- Folge: Ohne geeignete Synchronisation wird das Programm unbrauchbar



```

public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;

    public static void main(String[ ] args) throws InterruptedException {

Thread one = new Thread(new Runnable() {
        public void run() {
            a = 1;
            x = b;
        }
    });

Thread other = new Thread(new Runnable() {
        public void run() {
            b = 1;
            y = a;
        }
    });
    one.start(); other.start();
    one.join(); other.join();
    System.out.println("(" + x + "," + y + ")");
    }
}

```

Leicht zu verstehen wie (0,1),
(1,0) oder (1,1) rauskommen
kann
Jedoch auch möglich: (0,0)!

1.1.3 Das Java Memory Model



- **Definition:** Das Java Memory Model bestimmt, wie Threads über den Speicher interagieren
- **Es bestimmt :**
 - wie Variablen geschrieben und gelesen werden
 - wie Threads gestartet und beendet werden
 - wie Monitore geschlossen/ freigegeben werden
 - die Abarbeitungsreihenfolge der Befehle

1.2 happens-before Beziehungen



- JMM definiert eine Ordnung für alle Aktionen im Programm
 - Diese Ordnung garantiert: A sieht B, falls eine happens-before Beziehung zwischen B und A besteht
 - Besteht keine Beziehung, darf die JVM nach belieben den Ablauf der Befehle umordnen
-
- Diese Ordnung ist:
 - Reflexiv: A happens-before A
 - Transitiv: A happens-before B und B happens-before C
--> A happens-before C
 - Antisymmetrisch: A happens-before B und B happens-before A
--> A = B

1.2.1 Happens-before Beziehungen – Regeln 1



- **Programmreihenfolge:**
 - Es besteht eine happens-before Beziehung zwischen zwei Befehlen A und B, wenn B nach A im gleichen Thread kommt
- **Monitor lock:**
 - Es besteht eine happens-before Beziehung zwischen der Freigabe eines Monitor locks und einem späteren wiederabschließen
- **Volatile Variable:**
 - Es besteht eine happens-before Beziehung zwischen dem Schreiben auf eine volatile Variable und dem späteren Lesen
- **Thread-Start:**
 - Es besteht eine happens-before Beziehung zwischen dem Thread.start-Aufruf und jedem Befehl, der in dem Thread ausgeführt wird

1.2.1 Happens-before Beziehungen – Regeln 2



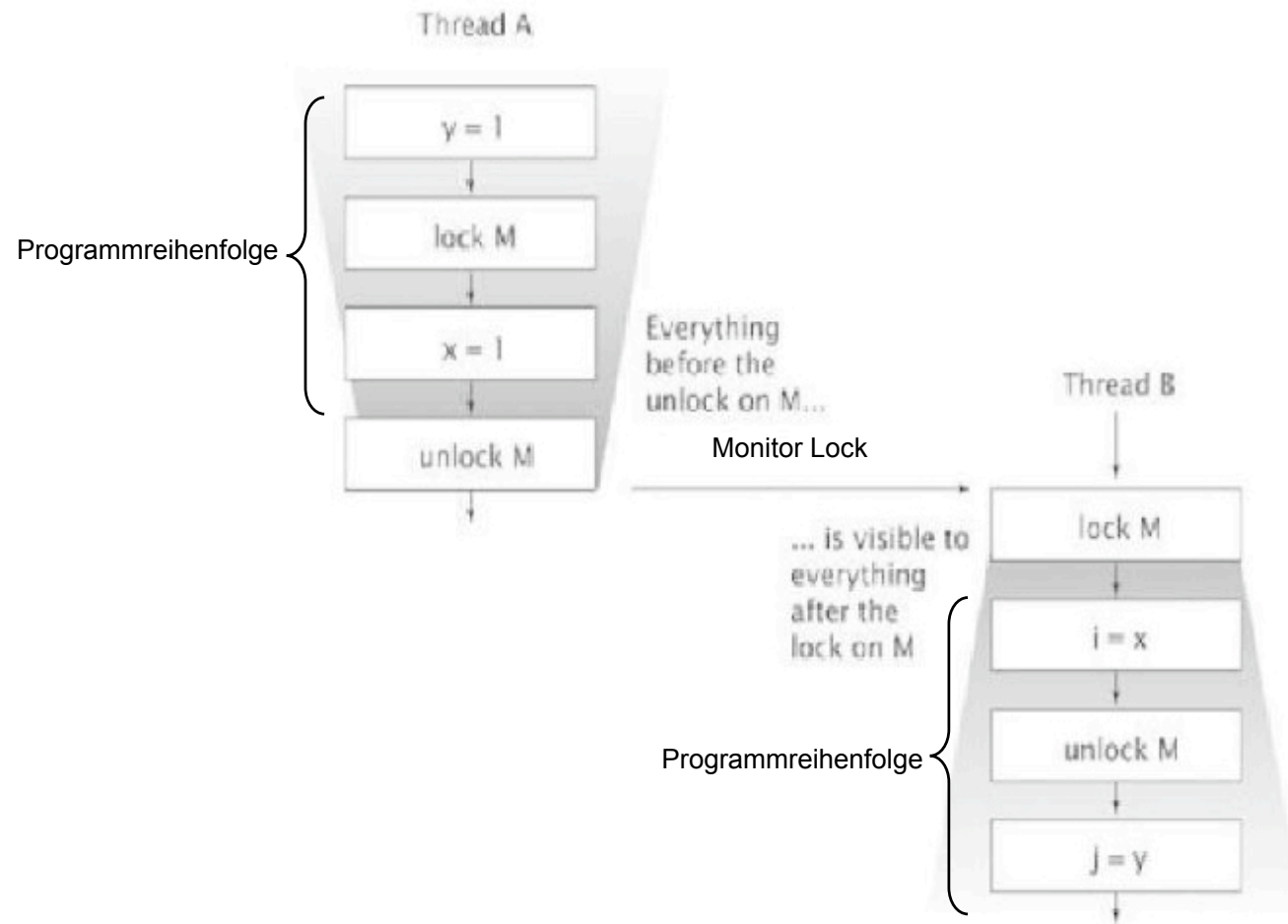
- **Thread-Beendigung:**
 - Alle Aktionen im Thread happens-before irgendein anderer Thread dessen Terminierung feststellt. Dies geschieht entweder wenn `thread.join` erfolgreich zurückkehrt oder `thread.isAlive` false liefert
- **Unterbrechung:**
 - Es besteht eine happens-before Beziehung zwischen dem Aufruf von „interrupt“ durch Thread A auf einen anderen Thread B und dem Bemerkten der Unterbrechung durch B
- **Finalizer:**
 - Zwischen dem Ende des Konstruktors und dem Start des Finalizers besteht eine happens-before Beziehung
- **Transitivität:**
 - Wenn A vor B und B vor C, dann A vor C

1.2.2 Huckepack



- **Idee:**
die happens-before-Relation ist so stark, dass man an diese weitere unsynchronisierte Methoden hängen kann
- **Ausführung:**
Eine in der Programmreihenfolge später kommende Methode im Thread A wird mit einer Methode des Threads B synchronisiert, so dass eine in Thread B folgende Methode auch auf alle Fälle später kommt.

1.2.2 Piggybacking on synchronization - Beispiel



```
private final class Sync extends AbstractQueuedSynchronizer {  
.....  
private V result;  
private Exception exception;  
  
void innerSet(V v) {  
    while (true) { ..... }  
    result = v;  
    releaseShared(0);  
    done();  
}  
  
    V innerGet() throws InterruptedException, ExecutionException {  
        acquireSharedInterruptibly(0);  
        .....  
        return result;  
    }  
}
```

1.3.1 Safe Publication



Publication über Liste:

Thread A stellt X in eine Blocking Queue; Danach holt Thread B das Objekt aus der Liste

- Veröffentlichung sicher, weil die Blocking-Queue synchronisiert ist
-> B erhält den Wert X garantiert auf dem neuesten Stand
- Nicht aber alle anderen Werte von Thread A

Happens-before- Verbindung:

Thread A und B teilen sich eine Variable X, die von einem Lock bewacht wird. A schreibt auf die Variable, danach liest B

- Veröffentlichung von X sicher
- B sieht alle von A vorher gemachten Änderungen (Transitivität)
- B sieht nicht sicher, was A nach dem Schreiben gemacht hat

1.3.2 Unsafe Publication



- Ohne happens-before wird die Publication unsicher
- Thread B sieht von Thread A erzeugtes Objekt anders als Thread A (out-of-date data)

Mit Ausnahme der unveränderlichen Objekte ist es nicht sicher ein Objekt zu benutzen, das von einem anderen Thread erzeugt wurde, es sei denn die happens-before Regeln wurden beachtet!

@NotThreadSafe

```
public class UnsafeLazyInitialization {  
    private static Resource resource;
```

```
    public static Resource getInstance() {  
        if(resource == null)  
            resource = new Resource(); //unsichere Publication  
        return resource;  
    }  
}
```

1.3.3 Safe initialization idioms



- oft sinnvoll, Initialisierungen nur bei Bedarf zu machen
→ Performance-Steigerung
- Vorsicht: Falsche Implementierung erzeugt Fehler (unsafe publication!)

@ThreadSafe

```
public class SafeLazyInitialization {  
    private static Resource resource;
```

```
    public synchronized static Resource getInstance() {  
        if(resource == null)  
            resource = new Resource();  
        return resource;  
    }  
}
```

1.3.4 Double-checked locking (DCL)



- Veraltete Technik, da sich bei frühen JVMs die Synchronisation negativ auf die Performance auswirkte
→ Idee von DCL: Synchronisation nur bei Bedarf

Technik:

- Prüfung: ohne Synchronisation
- Rückgabe, wenn vorhanden
- Wenn nicht, dann noch mal synchronisiert prüfen und gegebenenfalls Initialisieren

1.3.4 Double-checked locking (DCL)



- **Problem:**
 - Bei der 1. Prüfung kann es passieren, dass der Thread nur ein unvollständig initialisiertes Objekt sieht und dieses dann weiter gibt
- **Lösung:**
 - Objekt muss volatile sein
- Heute ist dieser Weg der Initialisierung überflüssig, da Lazy-Initialisation die gleichen Vorteile bringt wie DCL, aber leichter zu verstehen ist
- Andere Optimierungen im Programm sind wesentlich effektiver als DCL, weil die eigentlichen Gründe (langsame JVM, Synchronisation) weggefallen sind

@NotThreadSafe

```
public class DoubleCheckedLocking {  
    private static Resource resource;
```

```
    public static Resource getInstance() {  
        if(resource == null) {  
            synchronized (DoubleCheckedLocking.class) {  
                if (resource == null)  
                    resource = new Resource();  
            }  
        }  
        return resource;  
    }  
}
```

1.4 Initialization safety



- Initialization safety garantiert, dass als final deklarierte Objekte nach ihrer Initialisierung ohne Synchronisation geteilt werden können
 - Während der Initialisierung von X werden alle Objekte mit Referenzen zu X eingefroren
 - Initialisation safety verbietet jegliches reordering, das stören könnte
 - wenn ein Befehl ein Objekt initialisiert, das durch final deklarierte Variablen referenziert wird, unterliegt es nicht dem reordering

```
@ThreadSafe
public class SafeStates {
    private final Map<String, String> states;

    public SafeStates() {
        states = new HashMap<String, String>();
        states.put("alaska", "AK");
        states.put("alabama", "AL");
        states.put("wyoming", "WY");
        //etc.
    }

    public String getAbbreviation(String s) {
        return states.get(s);
    }
}
```