Seminararbeit

Java Thread Synchronisation

Angelika Brückl (0055060)

Inhaltsverzeichnis

1. Einleitung

Eine der Stärken der Programmiersprache Java ist die direkte Unterstützung von Threads. Threads erlauben die Bearbeitung mehrere Programmblöcke simuliert zeitparallel. So ist es möglich verschiede Aufgaben von Programmen auf Threads zu verteilen und damit quasi gleichzeitig zu bearbeiten, was *Multithreading* genannt wird. Die Synchronisation von Threads und von Zugriffen auf Daten erfolgt in Java durch so genannte *Monitore*.

2. Allgemeines über Threading

2.1. Was ist ein Thread?

Threads sind Teile eines normalen Prozesses. Prozesse sind über eigene Prozessräume voneinander abgeschottet. Alle Threads eines Prozesses werden im Unterschied dazu in demselben Adressraum ausgeführt. Threads sind also leicht-gewichtige Prozesse, die durch eine Kombination von shared memory und message passing miteinander kommunizieren können.

2.2. Wozu Threading?

Nacheinander aufgeschriebene Anweisungen werden üblicherweise sequentiell ausgeführt. In besonderen Fällen ist es jedoch sinnvoll davon abzuweichen:

1. Wenn die Programmausführung auf mehreren Prozessen verteilt wird, um eine Leistungssteigerung zu erzielen.
2. Um weitgehend unabhängige Vorgänge zu realisieren, die zeitlich (fast) beliebig ineinander greifen können.

Diese Ansätze gehen aus den folgenden Überlegungen hervor:
Wenn ein Rechensystem mehrere Prozessoren besitzt, könnte bei rein sequentieller Programmausführung nur einer von diesen arbeiten. Zur Steigerung der Rechenleistung (siehe Punkt 1) kann ein Programm in so viele Threads aufgeteilt werden, wie Prozessoren vorhanden sind. So können alle Prozessoren gleichzeitig arbeiten und die ganze Ausführungsdauer des Programms sinkt. Diese Threads sind nebenläufig zueinander, d.h. sie können parallel ausgeführt werden. Jeder unabhängig von dem anderen mit seiner eigenen Geschwindigkeit, solange nicht auf Daten oder bestimmte Ereignisse von Nachbar-Threads gewartet werden muss.
Es ist aber auch möglich ein Programm in mehrere Threads zu zerlegen, auch wenn nur ein Prozessor vorhanden ist (siehe Punkt 2). In diesem Fall führt das Betriebsystem die Threads *quasiparallel* aus. Genau genommen arbeiten die Threads dann zeitlich abwechselnd.
Grundsätzlich bedeutet Nebenläufigkeit, dass zu einem Zeitpunkt mehrere Threads existieren, die ausgeführt werden können, aber nicht müssen. Die tatsächliche Ausführung kann echt parallel oder quasiparallel erfolgen. Die Aufgabe von Java ist aber nicht die Verteilung von Threads auf Prozessoren.

2.3. Wozu Synchronisation von Threads?

Solange jeder Thread eigenständig für sich läuft, gibt es keine Probleme. Will er aber auf gemeinsam genutzte Ressourcen zugreifen, wird es problematisch. Es können zum Beispiel nicht zwei Threads gleichzeitig auf dieselbe Ressource schreiben.
Die zusammenhängenden Programmblöcke, die nicht unterbrochen werden dürfen und besonders geschützt werden müssen, nennt man kritische Abschnitte. Bei einem kritischen Abschnitt sollen die Operationen entweder ganz oder gar nicht ausgeführt werden. Dieser Codeabschnitt enthält nur atomare Operationen (siehe auch Kapitel 4 – Monitore).

3. Überblick – Threads in der JVM

Jede JVM unterstützt mehrere Threads konkurrent nebeneinander. Threads werden mit Hilfe der Klasse Thread oder Thread Group erzeugt, oder durch Implementierung des Interface Runnable. Mit Hilfe dieser Klassen oder dem Interface ist es möglich den Thread zu steuern mit den bekannten Befehlen, wie start(), run(), stop(). Wenn die start()-Methode aufgerufen wird, dann beginnt die Abarbeitung der run()-Methode dieses Threads.

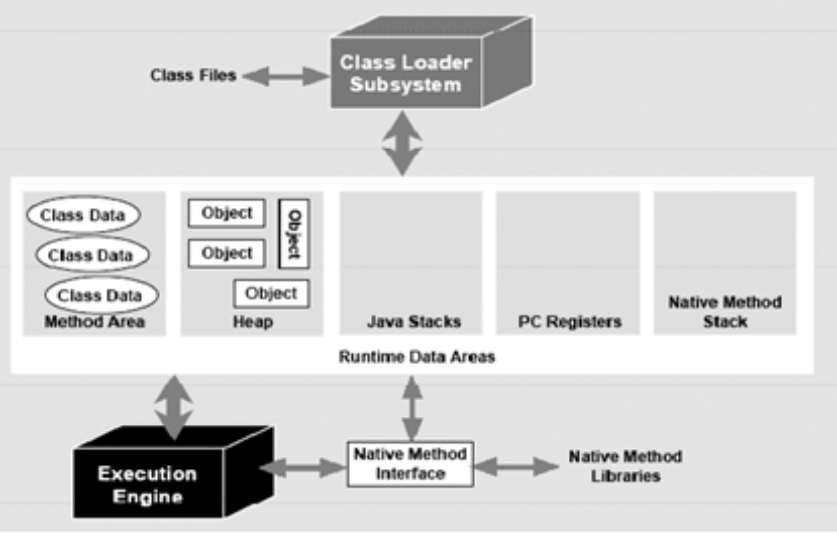

Abb. 1: Überblick: Java Virtual Machine

Zeitgleich wird mit dem Thread ein privater Stack dieses Threads angelegt. Dort werden alle privaten Variablen dieses Threads gespeichert. Andere Threads haben darauf keinen Zugriff. Den Heap und die Method Area müssen sich alle Threads teilen. Genau in diesen Bereichen muss dann die Synchronisation von den Threads erfolgen.

4. Monitore

Das Konzept des Monitors wurde von C.A.R. Hoare eingeführt und passt von der Konstruktion her sehr gut zur objektorientierten Programmierung. Mit Hilfe von Monitoren wird in Java Synchronisation von Threads unterstützt. Unter einem Monitor kann man sich ein Objekt vorstellen, das einen Thread blockieren und von der Verfügbarkeit einer Ressource benachrichtigen kann.
Um sich einen Monitor besser vorstellen zu können, kann die Gebäude-Anologie, wie sie Bill Venners in seinem Buch „Iside the Java Virtual Machine“ beschreibt, hilfreich sein:

Einen Monitor kann man sich wie ein Gebäude mit 3 Räumen vorstellen - einer Eingangshalle, einem Behandlungsraum und einem Warteraum.

Eingangshalle = *entry set*	Thread erreicht Anfang des gegenseitigen Ausschlusses
Behandlungsraum = *monitor region*	gegenseitiger Ausschluss wird behandelt
Warteraum = *wait set*	Wartepause bei Kooperation

Enthält der Behandlungsraum Daten, so hat der Thread exklusiven Zugriff vom Zeitpunkt des Eintretens bis zum Verlassen des Raumes.

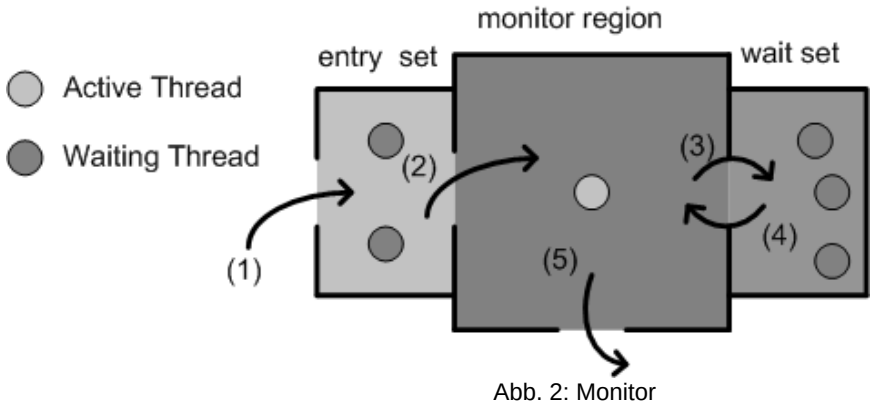

Abb. 2: Monitor

(1)	*Enter the entry set*	„Gebäude über Eingangshalle betreten“
(2)	*Acquire the monitor*	„Behandlungsraum betreten und benützen“
(3)	*Release the monitor and enter the wait set*	„Warteraum aufsuchen“
(4)	*Acquire again the monitor*	„Behandlungsraum wieder benutzen“
(5)	*Release and exit the monitor*	„Gebäude verlassen“

Kommt ein Thread außerhalb der *monitor region* an, wird er in das *entry set* platziert, praktisch in die Eingangshalle des Monitor-Gebäudes. Wartet kein Thread, bzw. besitzt kein Thread den Monitor, so kann er diesen betreten. Wenn der Thread fertig ist, dann verlässt er den Monitor und gibt ihn frei. Ist der Behandlungsraum jedoch besetzt, so muss der Thread im *entry set* warten. Wird der Monitor dann frei, muss er mit eventuell. anderen wartenden Threads konkurrieren, wer den Behandlungsraum betreten darf. Nur ein Thread kann gewinnen und den Monitor erwerben.

Die Codeabschnitte, die synchronisiert werden müssen – heißen *critical regions* (kritische Abschnitte). Es liegt ein kritischer Abschnitt vor, wenn der Code eine atomare Operation enthält, d.h. der Code soll ganz oder gar nicht ausgeführt werden. Kein anderer Thread darf den Monitor zwischenzeitlich benutzen. Es sind nur die Operationen *(1) enter the entry set* und *(5) release and exit the monitor* ausführbar und *(2) acquire the monitor* nur nach einer Operation und darauf *(5) realease and exit the monitor*.

In Java werden zwei Synchronisationsarten unterstützt mit Hilfe von Monitoren:

- Gegenseitiger Ausschluss (mutual exclusion)
- Kooperation

4.1. Gegenseitiger Ausschluss

Es wird verhindert, dass 2 oder mehr Prozesse gemeinsam auf Daten zur gleichen Zeit schreiben oder lesen. Dies wird mittels *Object-Locks* realisiert. Nach Erzeugen eines Objekts ist dessen Lock verfügbar. Sobald ein Thread einen kritischen Bereich erreicht, wird automatisch der dazugehörige Lock angefordert. Falls verfügbar, wird der Lock dem aufrufenden Thread zugeordnet und ist dann für andere Threads nicht mehr verfügbar. Andernfalls wird der Thread in die Warteliste des angeforderten Lock eingetragen. Bei Verlassen des kritischen Bereichs wird der Lock automatisch freigegeben. Falls Threads warten, erhält einer von ihnen den Lock. Welcher Thread diesen erhält ist nicht festgelegt.
Unterstützt das Laufzeitsystem kein Time Slicing, dann kann es passieren, dass ein Thread mit einer höheren Priorität den Prozessor die ganze Zeit beansprucht – „Monopolisierung der CPU“. Dann haben Threads mit niedrigeren Prioritäten wenig Chance die CPU zugeteilt zu bekommen.

Beispiel zum gegenseitigen Ausschluss

Mehrere Threads verwalten ein Internet-Bankkonto. Ein Thread, der mehr von dem Konto abheben möchte, als gegenwärtig aufgebucht ist, soll warten, bis ein oder mehrere andere Threads auf diesem Konto ausreichend eingezahlt haben.

```
public class Account {

	int bankBalance; ...

	//Geld abheben vom Konto
	public synchronized void DebitAcct (int amount) {
		while (bankBalance - amount) < 0) wait();
		bankBalance = bankBalance-amount; ...
	}

	//Geld einzahlen auf das Konto
	public synchronized void CreditAcct (int amount) {
		bankBalance = bankBalance+amount;
		notify(); ...
	}
}
```

4.2. Kooperation

Mehrere Threads arbeiten miteinander für ein gemeinsames Ziel. Dies wird unterstützt mittels wait() und notify() der Klasse object. Wait() wird von einem Prozess aufgerufen, wenn dieser mit der Exekution nicht fortfahren kann, so sollte ein anderer Prozess fortfahren: Notify() benachrichtigt andere Prozesse, dass sie nun mit der Ausführung fortfahren können.

Beispiel zu Kooperation

```
class T extends Thread {
	static int result;
	public void run() {
		...
	}
}
```

Ein Thread besitzt seine eigenen Variablen, etwa die Objektvariablen, kann aber auch statische Variablen nutzen.

In diesem Fall können verschiedene Thread-Objekte der Klasse T Daten austauschen, indem sie die Informationen in result ablegen oder daraus entnehmen. Threads können aber auch an einer zentralen Stelle eine Datenstruktur erfragen und dort Informationen entnehmen oder Zugriff auf gemeinsame Objekte über eine Referenz bekommen. Es gibt also viele Möglichkeiten, wie sich Threads – und damit potenziell parallel ablaufende Aktivitäten – Daten austauschen, also kooperieren, können.

Wenn eine Klasse eine oder mehrere synchronisierte Methoden besitzt, so hat jede Instanz dieser Klasse einen Monitor.

5. Thread Scheduling

Jeder Thread der JVM kann sich in folgendem der vier Zustände befinden.

⇨ **Initial**
Ein Thread-Objekt befindet sich in diesem Zustand vom Zeitpunkt an, an dem es erzeugt wurde, also wenn der Konstruktor erzeugt wird, bis zum Zeitpunkt, wo die start() – Methode von diesem Thread-Objekt aufgerufen wird.

⇨ **Runnable**
Ein Thread befindet sich in diesem Zustand, wenn die start() – Methode aufgerufen wurde. (Vergleich: Thread befindet sich in der Eingangshalle – *entry set*)

⇨ **Blocked**
Ein Thread, der geblockt wird und warten muss, bis ein bestimmtes Ereignis eingetreten ist. Die schlafenden und wartenden Threads werden da auch dazugezählt. (Vergleich: Thread ist im Warteraum – *wait set*)

⇨ **Exiting**
Ein Thread ist in diesem Zustand, wenn bei der run() – Methode ein return zurückgegeben wird, oder seine stop() – Methode aufgerufen worden ist. (Vergleich: Thread verlässt wieder das Monitor-Gebäude)

Es kommt öfters vor, dass sich mehrere Threads im Zustand *Runnable* befinden. Dann muss ein Thread ausgewählt werden, der dann der *Owner* des Monitors wird. Es stellt sich nun die Frage, wie dieser Thread ausgewählt wird.

Java verwendet einen fixen Preemptive-Priority-Scheduling-Algorithmus. Jedem Thread wird eine bestimmte Priorität zugewiesen, welche nur vom Programmierer verändert werden kann. Der Thread mit höherer Priorität wird als erstes ausgewählt von allen Threads, die im Zustand *Runnable* sind. Wenn ein Thread mit einer höheren Priorität den Monitor betritt, wird ein Thread mit niedrigerer Priorität, der gerade der *Owner* ist, unterbrochen. Der Thread mit der höheren Priorität ist dann der *Owner* des Monitors.

5.1. Scheduling von Threads mit gleichen Prioritäten

In den meisten Java-Programmen werden aber Threads mit gleichen Prioritäten auftreten. Wie werden diese dann gescheduled?

Man kann sich vorstellen, dass ein Monitor mit Hilfe von verketteten Listen implementiert wird. Jede Liste enthält Threads in einem der vier Zustände. Weiters kann ein Thread nur eine von elf Prioritäten haben. Also haben wir insgesamt vierzehn verkettete Listen (für jeden der 4 Zustände – 4 Listen und für jede Priorität – 11 Listen). In einer Prioritäten-Liste befin-

den sich immer nur Threads, die im Zustand *Runnable* sind, z.B. ein Thread der Priorität 7 und im Zustand *Runnable* ist dann in der Liste mit der Priorität 7. Wird der Thread geblockt, dann wird er in die Liste der Threads, die im Zustand geblockt sind eingefügt. Hier wird der Thread, der gerade *Owner* des Monitors war an das Ende der jeweiligen Liste angehängt, bzw. wird immer der Thread, der als erster der jeweiligen Liste ist ausgewählt. Das ist eine gängige Implementierung in der JVM, aber sie muss nicht so erfolgen. Hier wird aber kein Thread gleicher Priorität von einem anderem unterbrochen, wie es beim Round-Robin-Verfahren der Fall ist.

Es ist nicht vorgeschrieben, dass in der JVM das Round-Robin-Verfahren implementiert ist. Meistens ist es davon abhängig, ob ein Betriebsystem *time-slicing* unterstützt, was bei den „Nicht-Windows-Plattformen" eher nicht der Fall ist.
Das Round-Robin-Verfahren benützt einen internen Timer, der bestimmt, wann ein *Owner*-Thread unterbrochen wird und der nächste Thread der gleichen Priorität an der Reihe ist. Dieser führt dann seine Operationen aus, bis er geblockt wird, oder ein Thread mit einer höheren Priorität den Monitor betritt.

5.2. Implementierung von Java-Threads in Betriebssystemen

Viele Implementierungen der JVM erlauben dem Betriebssystem das Scheduling zu übernehmen. Die Betriebssysteme verwenden auch generell Priority-Based-Scheduling.

Dazu gibt es zwei grundlegende Modelle:

⇨ **Green-Thread Modell**
Green-Threads sind simulierte Threads innerhalb des Virtual-Machine-Prozesses. Sie werden in der Virtual Machine selbst realisiert.

⇨ **Native-Thread Modell**
Die Threads der JVM werden auf native Threads im Betriebsystem abgebildet. Dadurch ist die Skalierbarkeit auch deutlich besser. Ein Betriebssystem-Kernel auf einem Multiprozessor-System kann die Java-Threads auf verschiedene Prozessoren verteilen, wodurch die Threads echt parallel ablaufen. Auch das Scheduling kann direkt im Betriebssystem erfolgen.

Konkrete Beispiele:

Windows NT/2000/XP
Hier werden Java-Threads auf Threads im Betriebsystem abgebildet, die innerhalb des Virtual-Maschine-Prozesses ablaufen. Im Task-Manager kann man gut die gestarteten Prozesse und Threads beobachten. Beim Starten eines Java-Programmes, sieht man, dass sich die Anzahl der Prozesse um Eins erhöht, die Anzahl der Threads nimmt dagegen stärker zu. Dies ist auch der Fall, wenn man im Programm selber keine Threads programmiert hat, weil die JVM eigene System-Threads für Aufgaben wie Garbage Collection startet.

Solaris
Hier werden seit der JDK-Version 1.2 die nativen Threads eingesetzt. In den Vorversionen machte die Solaris-Virtual Machine von Green Thread Gebrauch.

Linux
Hier werden Threads seit der JDK-Version 1.3 mit Hilfe von abgespalteten Tochterprozessen realisiert. Zuvor verwendete man auch hier Green Threads. Dadurch, dass für Java-Threads komplette Prozesse erzeugt werden, kann die Erzeugung einer großen Anzahl neuer Threads länger dauern als bei Betriebssystemen, auf denen native Threads eingesetzt werden.

5.3. Deadlocks

Ein Deadlock (auch *Verklemmung* genannt) ist ein Zustand von Prozessen, bei dem mindestens zwei Prozesse untereinander auf Betriebsmittel warten, die dem jeweils anderen Prozess zugeteilt sind.

Auf ein Java-Programm übertragen könnte ein Deadlock zustande kommen, wenn sich zwei Monitore in den Zugangsmethoden gegenseitig aufrufen können. In diesem Fall kann es vorkommen, dass ein Thread den Monitor A betritt und ein anderer den Monitor B. Wenn nun in der Zugangsmethode von A eine Zugangsmethode von B aufgerufen wird und in B eine von A, so wartet der erste Thread darauf, dass der zweite den Monitor B verlässt, dieser wartet jedoch darauf, dass der erste den Monitor A verlässt.

Um derartige Zustände zu vermeiden, sollten inhaltlich zusammengehörige Funktionen, die synchronisiert ablaufen müssen, auch in einem einzigen Monitor abgehandelt werden. Aufrufe von Zugangsmethoden aus anderen Monitoren heraus sind generell gefährlich.

Ein weiteres Problem im Zusammenhang mit Monitoren ist die Methode Thread.stop(). Der stop()-Aufruf bewirkt, dass der angesprochene Thread sofort alle Monitore freigibt, die er besitzt, und zwar auch dann, wenn er eine Zugangsmethode noch nicht bis zum Ende ausgeführt hat. Dadurch kann es passieren, dass ein gestoppter Thread einen Monitor in einem inkonsistenten Zustand verlässt, was beim nächsten Thread, der den Monitor betritt, zu großen Problemen führen kann. Weil dieser Problematik kaum Abhilfe zu schaffen ist, wurde Thread.stop() seit Version 1.2 verworfen.

Dasselbe Schicksal ereilte die Methoden Thread.suspend() und Thread.resume(), weil ein Thread beim Suspendieren alle Monitore behält, die er besitzt. Wenn der Thread, der den resume()-Aufruf ausführt, zuvor den Monitor betreten muss, den der suspendierte Thread besitzt, resultiert ein Deadlock.

Der Vollständigkeit halber sei an dieser Stelle noch der Modifier volatile erwähnt, mit dem Datenelemente versehen werden sollten, auf die unsynchronisiert zugegriffen wird.

6. Synchronisation auf Bytecode-Ebene

Mit dem Schlüsselwort **synchronized** werden in Java Monitore gekennzeichnet. Es gibt 2 Möglichkeiten Monitore in einem Programm zu kennzeichnen. Mit Hilfe von *synchronized statements* oder von *synchronized methods*.

6.1. Synchronized Statements

S*ynchronized statements* bestehen aus dem Schlüsselwort **synchronized** und einer Objektreferenz.

Beispiel:

```
class KitchenSync {

       private int[] intArray = new int[10];

       void reverseOrder() {
              synchronized(this) {
                     int halfWay = intArray.length/2;
                     for (int i=0; i< halfWay; ++i) {
                            int upperIndex - intArray.length-1-i;
                            int save = intArray[upperIndex];
```

```
                        intArray[upperIndex] = intArray[i];
                        intArray[i] = save;
                    }
                }
            }
}
```

Der Block, der durch das synchronized (this) { … } eingeschlossen ist, kann erst dann ausgeführt werden, wenn der Lock für das Objekt (this) erhalten wurde. Im Bytecode werden zwei Opcods *monitorenter* und *monitorexit* für die Synchronisation von Blöcken verwendet.

`monitorenter`
Dieser Befehl kennzeichnet den Beginn des synchronisierten Code Blockes in einer Bytecodesequenz. Was geschieht beim Aufruf dieses Befehles?

- ⇨ Anfordern eines Locks auf objectref (-> Objektreferenz)
- ⇨ Durchführung Operation 1) *Enter the entry set*
- ⇨ Durchführung Operation 2) *Acquire the monitor*, falls kein
- ⇨ Thread *Owner* dieses Monitors ist
  monitorzähler = monitorzähler + 1
- ⇨ Sonst Einordnung ins Entry Set

`monitorexit`
Dieser Befehl befindet sich am Ende des synchronisierten Code Blockes in einer Bytecodesequenz. Was geschieht beim Aufruf dieses Befehles?

- ⇨ objektref wird wieder freigegeben
- ⇨ *monitorzähler = monitorzähler -1*
- ⇨ Durchführung Operation 5) *Release and exit the monitor*, falls
- ⇨ *monitorzähler* zu Null wird
- ⇨ Sonst Durchführung des Operationen 2) *Acquire the monitor* oder
  4) *Acquire again the monitor* abhängig von den *notify*-Bedingungen

Tritt eine Exception in dem synchronisierten Code Block auf, dann sorgt ein monitorexit im catch-Block, dass das Objekt freigegeben wird.

6.2. Synchronized Methods

Um eine Methode zu synchronisieren braucht man nur das Schlüsselwort synchronized als einen der Method-Qualifiers hinzuzufügen.

Beispiel:

```
class HeatSync {

      private int[] intArray = new int[10];

      synchronized void reverseOrder() {
            int halfWay = intArray.length/2;
                  for (int i=0; i< halfWay; ++i) {
                        int upperIndex - intArray.length-1-i;
                        int save = intArray[upperIndex];
                        intArray[upperIndex] = intArray[i];
                        intArray[i] = save;
                  }
            }
      }
      //...
}
```

Die JVM braucht keine speziellen Opcodes um die synchronisierte Methode aufzurufen, oder zurückzugeben. Stößt die JVM auf eine Referenz einer Methode, kann sie feststellen, ob es sich um eine synchronisierte Methode handelt, oder nicht. Bei einer Methode für eine Instanz, fordert die JVM einen Lock im Zusammenhang mit dem Objekt an mit dem die Methode aufgerufen wird. Handelt es sich um eine Methode einer Klasse, dann fordert die JVM den Lock im Zusammenhang dieser Klasse an. Wurde die synchronisierte Methode fertig ausgeführt, oder zurückgegeben, dann gibt die JVM den Lock wieder frei.

Ein Monitor kann von einem Thread beliebig oft aufgerufen werden.

7. Modell: Wie könnte die Koordination von Thread und gemeinsamen Speicher erfolgen? *nach James Gosling, Bill Joy und Guy Steele.*

7.1. Definitionen und Regeln

Hauptspeicher: gemeinsam genutzt von allen Threads zur Speicherung von Variablen (Klassenvariablen, Instanzenvariablen, Komponenten in Arrays)

Arbeitsspeicher: jeder Thread hat eine Arbeitskopie „seiner“ Variablen (Hauptspeicher hat die Master-Kopie)

Lock (Schloss): jedes Objekt hat sein eigenes Schloss Threads konkurrieren um den Erwerb von Locks

Execution Engine: führt Byte-Befehlsfolge aus

Aktionen des Threads auf den Speicher:

use überträgt den Inhalt der Arbeitskopie einer Variablen des Threads an die Execution Engine des Threads

assign überträgt den Wert der Execution Engine des Threads in die Arbeitskopie des Threads

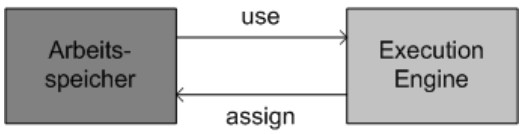

Abb. 3: Interaktion Arbeitsspeicher und Execution Engine

load überträgt einen Wert vom Hauptspeicher (Master-Kopie) durch eine vorgängige read Aktion in die Arbeitskopie des Threads

store überträgt den Inhalt der Arbeitskopie des Threads zum Hauptspeicher für eine spätere write Aktion

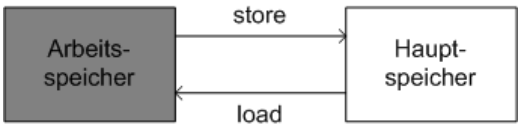

Abb. 4: Interaktion Arbeitsspeicher und Hauptspeicher

lock Thread beansprucht ein bestimmtes Schloss
(Beginn - Synchronisation zwischen Thread und Hauptspeicher

unlock Thread gibt ein bestimmtes Schloss frei
(Ende - Synchronisation zwischen Thread und Hauptspeicher)

Aktionen des Hauptspeichers:

read Überträgt Inhalt der Master-Kopie einer Variablen in die Arbeitskopie des Threads für eine spätere load Aktion

write überträgt einen Wert vom Arbeitsspeicher des Threads vermittels einer store Aktion in die Master-Kopie einer Variablen im Hauptspeicher

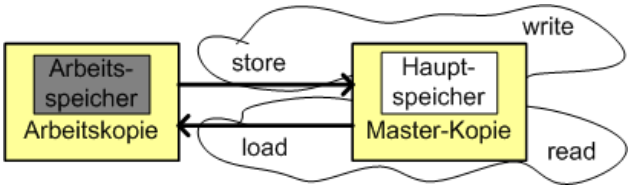

Abb. 5: Interaktion Arbeitsspeicher und Hauptspeicher

Regeln zu diesen Aktionen

- ⇨ Alle Thread-Aktionen, wie use, assign, load, store, lock, unlock und Speicher-Aktionen, wie read, write, lock, unlock sind atomar.
- ⇨ Alle Aktionen irgendeines Threads, eines Speichers für irgendeine Variable, eines Speichers für irgendeinen Lock sind total geordnet.
- ⇨ Threads kommunizieren nicht direkt miteinander, sondern nur über den gemeinsamen Speicher.
  Eine lock- oder unlock-Aktion wird gemeinsam von Thread und Hauptspeicher ausgeführt
  Nach einem load des Threads folgt immer ein read des Hauptspeichers
  Nach einem write vom Hauptspeicher folgt immer ein store vom Thread
  Alle diese Beziehungen sind transitiv, d.h. Wenn einer Aktion, Aktion B folgen muss und wenn vor Aktion die Aktion C stattfinden soll, dann muss die Aktion C vor der Aktion A ausgeführt werden.

Um das ganze ein bisschen anschaulicher zu machen 2 Beispiele:

7.2. Beispiel 1 – Nicht synchronisiertes Tauschen

Gegeben ist eine Klasse mit den Klassen-Variablen a und b und den Methoden hier() und da(). Angenommen es sind zwei Threads erzeugt worden und einer ruft die Methode hier() auf und der andere die Methode da(). Was wird passieren?
Nehmen wir zum Beispiel den hier-Thread. Grundvoraussetzung ist, dass dieser Thread ein use auf b durchführen kann, gefolgt von einem assign auf a. Die erste Operation kann aber kein use auf b sein. Also wird fürs erste einmal eine load-Aktion gebraucht, welche eine read-Aktion auf b vom Hauptspeicher voraussetzt. Wenn der Thread die Variable a speichern will, folgt nach der store-Aktion eine write-Aktion. Für den da-Thread gelten die gleichen Bedingungen.
Die Aktions-Reihenfolge würde dann wie folgt ablaufen.

```
class Beispiel {
  int a = 1, b = 2;
  void hier() {
    a = b;
  }
  void da() {
    b = a;
  }
}
```

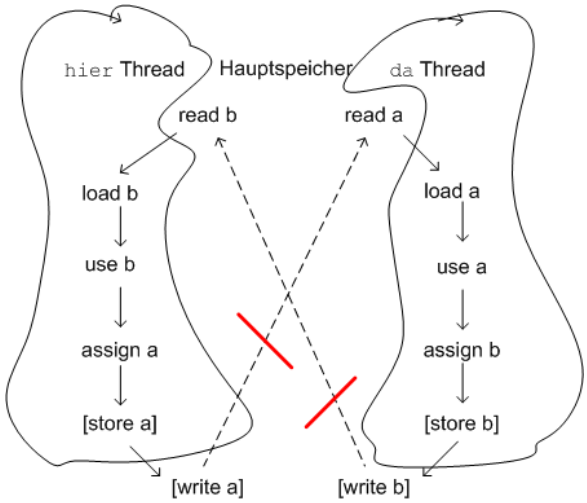

Abb. 6: Aktions-Reihenfolge zu Beisp. 1

Ein Pfeil von Aktion A zu Aktion B gibt an, dass die Aktion A der Aktion B vorausgeht. In welcher Reihenfolge werden jetzt die Aktionen vom Hauptspeicher gemacht? Die einzigen Kombinationen, die nicht möglich sind, wenn einer write-Aktion von a eine read-Aktion auf a vorausgeht und einer write-Aktion von b eine read-Aktion auf b vorausgeht. Dies ist deshalb nicht möglich, weil es eine Regel gibt, die verbietet, dass Aktionen sich selber folgen dürfen. Hier würde bei der Abarbeitung der Aktionen ein Kreis einstehen, was zur Folge hätte, dass die Aktionen sich selber folgen.

Folgende Operationen sind möglich:	**Ergebnis:**
• write a → read a, read b → write b	ha=2, hb=2, ma=2, mb=2, ya=2, yb=2
• read a → write a, write b → read b	ha=1, hb=1, ma=1, mb=1, ya=1, yb=1
• read a → write a, read b → write b	ha=2, hb=2, ma=2, mb=1, ya=1, yb=1
	ha, hb Arbeitskopien von hier-Thread da, db Arbeitskopien von da-Thread ma, mb Masterkopien im Hauptspeicher. Anfangswert: ma=1, mb=2

Das Ergebnis im Hauptsspeicher kann also folgendermaßen aussehen:

- b wurde in a hineinkopiert und a wurde in b hineinkopiert.
- a wurde in b hineinkopiert und b wurde in a hineinkopiert.
- Die Werte von a und b wurden vertauscht.

Keines der Ergebnisse trifft wahrscheinlicher ein als ein anderes.

7.3. Beispiel 2 - Synchronisiertes Tauschen

Hier wird dieselbe Klasse verwendet wie beim Beispiel 1, nur die beiden Methoden hier() und da() werden synchronisiert mit Hilfe dem Schlüsselwort synchronized.
Es werden wieder 2 Threads gestartet – einer ruft die Methode hier() auf und der andere die Methode da(). Ruft jetzt ein Thread die Methode hier() auf, so muss er einen Lock für diese anfordern. Der Lock wird auf das Klassen-Objekt der Klasse Syn gemacht. Dann können wie gewohnt die Aktionen ausgeführt werden. Nachdem der Rumpf der Methode ausgeführt wurde, wird durch eine unlock-Aktion das Objekt wieder freigegeben. Wie auch beim vorhergehenden Beispiel wird eine load-Aktion auf b gemacht, der eine read-Aktion auf b vorausgeht im Hauptspeicher. Da eine load-Aktion auf die lock-Aktion folgt, muss dieser load-Aktion eine read-Aktion vorausgehen, die auch nur nach der lock-Aktion folgen kann. Die write-Aktion muss der unlock-Aktion vorausgehen. Für den da-Thread gelten die gleichen Bedingungen.
Die Aktions-Reihenfolge würde dann wie folgt ablaufen.

```
class Syn {
  int a = 1, b = 2;
  synchronized void hier(){
    a = b;
  }
  synchronized void da(){
    b = a;
  }
}
```

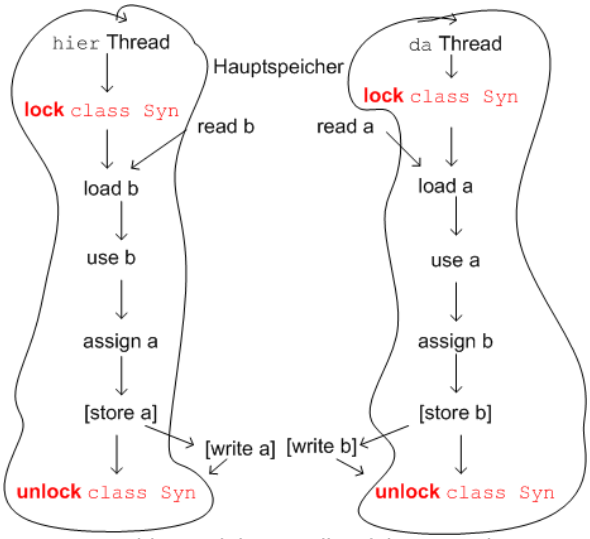

Abb. 7: Aktions-Reihenfolge zu Beisp. 2

Bei diesem Beispiel ist es nicht möglich, dass z.B. der hier-Thread Aktionen ausführt, wenn vorher der da-Thread eine Lock-Aktion gemacht hat und das Syn-Objekt noch nicht freigegeben worden ist durch eine unlock-Aktion.

Folgende Operationen sind möglich:	**Ergebnis:**
• write a → read a, read b → write b	ha=2, hb=2, ma=2, mb=2, ya=2, yb=2
• read a → write a, write b → read b	ha=1, hb=1, ma=1, mb=1, ya=1, yb=1
	ha, hb Arbeitskopien von hier-Thread da, db Arbeitskopien von da-Thread ma, mb Masterkopien im Hauptspeicher.

| Anfangswert: ma=1, mb=2

Das Ergebnis im Hauptsspeicher kann also nur folgendermaßen aussehen:

- b wurde in a hineinkopiert und a wurde in b hineinkopiert.
- a wurde in b hineinkopiert und b wurde in a hineinkopiert.

8. Literaturquellen

Bill Venners - Inside the Java 2 Virtual Machine, McGraw Hill, 1999

Frank Yellin, Tim Lindholm - The Java Virtual Machine Specification, Addison-Wesley 1997

Klaus Echtle, Michael Goedicke – Lehrbuch der Programmierung mit JAVA

Daniel Becker, Data Becker – JAVA – Das Grundlagenbuch – Das Fundament professioneller Java-Programmierung

Scott Oaks, Henry Wong, Java Threads, 2nd edition
http://arbow.itocean.net/resource/O'Reilly%20-%20Java%20Threads%20-%202ed.pdf

Stefan Middendorf, Reiner Singer, Jörn Heid: Programmierhandbuch und Referenz für die Java-2-Plattform, Standard Edition, 3. Auflage 2002
http://www.dpunkt.de/java//index.html