4 Elementare Synchronisationsmechanismen

Um Zugriffe auf gemeinsame Ressourcen zu kontrollieren, kann das Konzept des gegenseitigen Ausschlusses verwendet werden. Java stellt hierzu für jedes Objekt einen Lock- bzw. Sperrmechanismus zur Verfügung.

4.1 Schlüsselwort synchronized

Alle Java-Objekte, sowohl herkömmliche Instanzen als auch Klassenobjekte, besitzen eine implizite Sperre bzw. einen Lock. Den Zugriff darauf erhält man durch die Verwendung des Schlüsselworts `synchronized`.

Hinweis

Man kann mathematisch zeigen, dass eine gute Lösung für den gegenseitigen Ausschluss vier Bedingungen genügen muss [48]:

1. In einem kritischen Abschnitt darf sich zu jedem Zeitpunkt höchstens immer nur ein Thread befinden.
2. Es dürfen keine Annahmen über die zugrunde liegende Hardware (Clock, CPU-Anzahl etc.) gemacht werden.
3. Ein Thread darf andere Threads nicht blockieren, außer er ist in einem kritischen Bereich.
4. Es muss sichergestellt sein, dass ein Thread nicht unendlich lange warten muss, bis er in den kritischen Bereich eintreten kann.

4.1.1 Synchronized-Methoden

Jedes Objekt bzw. jede Klasse besitzt genau eine Sperre. Wird eine Methode mit dem Schlüsselwort `synchronized` deklariert, müssen alle Threads erst die Sperre erwerben, bevor sie die Methode ausführen dürfen. Bei

`synchronized`-Instanzmethoden ist es die Sperre des zugehörigen Objekts. Bei `synchronized`-Klassenmethoden kommt die Sperre des `class`-Objekts (*Klassenname*`.class`) zum Einsatz.

Beim Aufruf einer `synchronized`-Methode wird zuerst versucht, die Sperre anzufordern. Ist sie frei, erhält sie der aktuelle Thread und er führt die Methode aus. Ist sie belegt, wird er blockiert. Da es möglich ist, dass mehrere Threads dieselbe Sperre erwerben möchten, werden die blockierten Threads in einem Warteraum gesammelt. Mit dem Verlassen der Methode kann die Sperre freigegeben werden. Danach kann sie von einem blockierten Thread erworben werden. Welcher das sein wird, ist nicht festgelegt.

Die Sperre ist *reentrant*. Ein Thread kann, nachdem er die Sperre erlangt hat, sie ohne zu warten erneut anfordern. Dabei wird intern ein Zähler verwendet, der sicherstellt, dass der Thread die Sperre wieder genauso oft freigeben muss, wie er sie erhalten hat.

Als Beispiel betrachten wir einen Modulo-Zähler (Codebeispiel 4.1). Die Methode `increment` zählt den Counter hoch (❶) und `decrement` herunter (❷). Die Methode `getValue` liefert den aktuellen Wert (❸). Da sie alle `synchronized` sind, können sie nur von einem Thread ausgeführt werden. Außerdem ist die Sichtbarkeit von `count` gewährleistet. Beim Eintreten und Verlassen einer `synchronized`-Methode wird der Cache aktualisiert bzw. zurückgeschrieben (analoges Verhalten wie bei einem `volatile`-Zugriff).

```
class ModuloCounter
{
  private int count = 0;
  private final int mod;
  public ModuloCounter(int mod)
  {
    this.mod = mod;
  }
  public synchronized void increment()                    ❶
  {
    count = (count + 1)%mod;
  }
  public synchronized void decrement()                    ❷
  {
    count = (count - 1 + mod)%mod;
  }
  public synchronized int getValue()                      ❸
  {
    return count;
  }
}
```

Codebeispiel 4.1: Eine Thread-sichere Klasse

Da jedes Counter-Objekt eine eigene Sperre besitzt, ist es möglich, dass zwei Threads gleichzeitig `synchronized`-Methoden betreten können, wenn die Aufrufe auf verschiedenen Objekten (verschiedenen Sperren) erfolgen.

Klassenmethoden können ebenfalls mit `synchronized` gekennzeichnet werden. Gesperrt wird jetzt das der Klasse zugeordnete `class`-Objekt (z. B. `String.class`, `Integer.class` usw.). Dies bedeutet, dass zu jedem Zeitpunkt höchstens eine `static synchronized`-Methode einer Klasse ausgeführt werden kann.

4.1.2 Synchronized-Blöcke

Ein Thread kann auch explizit die Sperre eines beliebigen Objekts `obj` erhalten, wenn er in einen Block folgender Form eintritt:

```
synchronized(obj)
{
  // kritischer Abschnitt
}
```

Somit haben die beiden nachstehenden Versionen die gleiche Wirkung[1]:

```
public synchronized void f()
{
  kritischer Abschnitt
}

// Äquivalente Formulierung
public void f()
{
  synchronized(this)
  {
    // kritischer Abschnitt
  }
}
```

`Synchronized`-Blöcke können eingesetzt werden, wenn unabhängige Methoden geschützt werden sollen. Unabhängig bedeutet hier, dass sie auf disjunkte Datenbereiche zugreifen. Wären im Codebeispiel 4.2 beide Methoden `synchronized`, so könnte immer nur eine der Methoden aktuell von einem Thread ausgeführt werden, da dieselbe Sperre benutzt wird.

[1]Die beiden Versionen sind zwar semantisch äquivalent, besitzen aber verschiedene Laufzeiten. Die Lösung mit der `synchronized`-Methode kann im Bytecode effizienter umgesetzt werden als die mit dem `synchronized`-Block.

```
public class LockDemo
{
  private Object lock1 = new Object();
  private Object lock2 = new Object();

  public void method1()
  {
    synchronized (lock1)
    {
      // benutzt Attribute a,b,c
    }
  }

  public void method2()
  {
    synchronized (lock2)
    {
      // benutzt Attribute x,y,z
    }
  }
}
```

Codebeispiel 4.2: Einsatz von `synchronized`-Blöcken

4.1.3 Beispiel: Thread-sicheres Singleton

Codebeispiel 4.3 zeigt eine Implementierung des *Singleton*-Patterns mit verzögerter Initialisierung (*lazy instantiation*).

```
class Singleton
{
  private static Singleton instance;

  private Singleton()
  {
    // komplizierte Initialisierung
  }

  public static synchronized Singleton getInstance()
  {
    if (instance == null)
    {
      instance = new Singleton();
    }
    return instance;
  }

  // weitere Objektmethoden
 }
```

Codebeispiel 4.3: Ein Singleton mit *lazy instantiation*-Verhalten

Der Zugriff auf die Singleton-Instanz ist hier durch `synchronized` geschützt. Damit wird sichergestellt, dass nur ein Objekt erzeugt wird. Im Bytecode wird das Schlüsselwort `synchronized` durch einen Block mit dem Befehl `monitorenter` zu Beginn und `monitorexit` am Ende ersetzt. Die JVM verwendet dann die plattformabhängigen Lock-Mechanismen, um diese Sperre zu realisieren. Die Verwendung von `synchronized` ist aber auf jeden Fall mit einem Mehraufwand verbunden.

Eine Optimierung der Zugriffszeit ist das sogenannte *Double-Checked-Locking-Idiom*, bei dem `synchronized` nur bei der Erzeugung der Instanz eingesetzt wird[2]. Der lesende Zugriff auf ein bereits erzeugtes Singleton muss nicht mehr geschützt werden. Codebeispiel 4.4 zeigt die Implementierungsvariante.

```
private static volatile Singleton instance;                    ❶

public static Singleton getInstance()
{
  if (instance == null)                                        ❷
  {
    synchronized(Singleton.class)                              ❸
    {
      if (instance == null)
        instance = new Singleton();
    }
  }
  return instance;                                             ❹
}
```

Codebeispiel 4.4: Ein Singleton mit *Double-Checked-Locking*

Hier wird der Zugriff `getInstance` nicht geschützt, sodass er schnell ist, wenn das Objekt bereits erzeugt wurde. Muss das Objekt angelegt werden, muss sichergestellt sein, dass nur ein Thread den Konstruktor aufruft. Es ist eine zweite Abfrage notwendig (❸) (daher der Name Double-Checked), da es theoretisch möglich ist, dass ein Thread nach der ersten Abfrage (❷) verdrängt wurde. Die Variable `instance` muss als `volatile` deklariert sein (❶), weil auch außerhalb eines `synchronized`-Blocks auf sie zugegriffen wird (❹).

Auch wenn das Singleton-Objekt bereits existiert, wird beim Aufruf von `getInstance` zweimal ein *refresh* auf dem Cache ausgeführt, da zweimal auf die `volatile`-Variable zugegriffen wird (❷,❹). Durch Einführung einer lokalen Variablen kann man einen dieser Zugriffe vermeiden (siehe [5]). In

[2] Das *Double-Checked-Locking-Idiom* kann generell bei der Implementierung von *lazy instantiation* eingesetzt werden.

der Praxis sollte bei Java auf dieses Idiom verzichtet werden, da eine bessere Lösung existiert (siehe Praxistipp).

Praxistipp

Neben dem oben gezeigten Thread-sicheren Lazy-Instantiation-Singleton gibt es auch eine Möglichkeit ohne explizite Synchronisierung. Die Idee ist, die Instanziierung mit dem Klassenladen zu verknüpfen. Hierzu wird eine Hilfsklasse eingeführt, die erst beim Zugriff auf `getInstance` geladen wird (vgl. [16]). Der folgende Codeausschnitt zeigt die Idee:

```
public class Singleton
{
  // Holder-Klasse für Singleton
  private static class SingletonHolder {
     public static final Singleton instanceHolder = new Singleton();
  }

  public static Singleton getInstance()
  {
    // Beim ersten Aufruf wird die Klasse
    // SingletonHolder geladen
    return SingletonHolder.instanceHolder;
  }

  private Singleton()
  {
    super();
    ...
  }
  ...
}
```

Eine weitere Thread-sichere Variante basiert auf der Benutzung eines `enum`-Typs (siehe [5]).

4.1.4 Monitorkonzept bei Java

Schon in den 70er-Jahren haben Hansen [20] und Hoare [23] das Monitorkonzept eingeführt, um Multithreading sicher zu machen. Ein *Monitor* wäre auf Java übertragen eine spezielle Klasse mit folgenden Eigenschaften:

- Alle Daten der Klasse müssen `private` deklariert sein.
- Nur ein Thread kann zu jedem Zeitpunkt in einem Monitor aktiv sein, d.h., jeder Monitor hat eine Sperre.

- Die Sperre kann mit einer beliebigen Anzahl von Bedingungen (siehe Kapitel 5) verwendet werden.
- Es ist die Aufgabe der VM, den wechselseitigen Ausschluss der Monitoreingänge zu garantieren.

Dieses allgemeine Konzept wird in Java nicht gänzlich übernommen. Es gibt zwei wichtige Unterschiede:

1. Attribute einer Klasse müssen bei Java nicht `private` sein.
2. Nicht alle Methoden müssen als `synchronized` deklariert sein.

Das führt zum unsicheren Umgang, wie Hansen in seinem Artikel *Java's Insecure Parallelism* [19] kritisiert.

4.2 Fallstricke

- **Non-volatile-Zugriff auf gemeinsam benutzte Daten**
  Da ein lesender Zugriff auf eine `int`-Variable atomar ist, könnte man auf die Idee kommen, im Codebeispiel 4.1 das Schlüsselwort `synchronized` bei `getValue` wegzulassen (vgl. Codebeispiel 4.5).

```
class ModuloCounter
{
  private int count = 0;
  private final int mod;

  public ModuloCounter(int mod) { .... }

  public synchronized void increment()  { ... }
  public synchronized void decrement()  { ... }

  public int getValue()  // Falsch
  {
     return count;
  }
}
```

Codebeispiel 4.5: Eine inkorrekte Implementierung eines ModuloCounters

Die Methoden `increment` und `decrement` lösen zwar am Ende jeweils einen *flush* des Caches aus, sodass der aktuelle Wert von `count` in den Hauptspeicher übertragen wird. Beim Aufruf von `getValue` wird aber kein *refresh* des Caches des Aufrufers durchgeführt, sodass der Inhalt von `count` nicht unbedingt aktuell ist. Möchte man das `synchronized` bei `getValue` einsparen, so muss `count` mit `volatile` deklariert werden.

- **Gemeinsam benutzte Daten sind nur partiell geschützt**
  Dieser Fehler kommt aufgrund der Abweichung vom Monitorkonzept vor. Es existieren verschiedene Varianten:

  - Da nicht alle Daten immer als `private` deklariert sind, ist zum Beispiel der folgende Code nicht sicher:

```
class Unsafe
{
  int commonData;  // public, protected sind auch schlecht

  public synchronized void inc()
  {
    commonData++;
  }
}
```

    Ein anderer Thread kann durch den direkten Zugriff die Variable `commonData` lesen bzw. manipulieren, während ein anderer sie gerade ändert (nicht atomare Operation).

  - Wenn nicht alle Methoden, die mit einer gemeinsamen Variablen arbeiten, `synchronized` sind, kann sehr leicht ein Fehler eintreten:

```
class Unsafe
{
  private int commonData;

  public synchronized void inc()
  {
    commonData++;
  }

  public void doIt()
  {
    // Methode mit Änderung von commonData
    // oder mit einer Anweisung wie
    // if (commonData > 10)
  }
}
```

  - In der Praxis ist die folgende Fehlervariante auch zu beobachten: Eine `synchronized`-Methode ruft eine normale `public`-Methode eines anderen Objekts auf, das eventuell nebenläufig modifiziert werden kann. Das bedeutet, dass es nicht ausreichend geschützt ist.

  - Alle Methoden der Klasse `Vector` von Java sind `synchronized` deklariert. Das führt häufig zu der falschen Annahme, dass der Umgang mit einem `Vector`-Objekt Thread-sicher ist. Im folgenden Beispiel

```
// Attribut der Klasse
Vector vec = ...;
...

// in irgendeiner Methode
for (int i = 0; i < vec.size(); i++)
{
  // modifiziere vec
}
```

kann zwischen den Durchläufen ein anderer Thread `vec` verändern. Möchte man sicher über den Vektor traversieren, muss das ganze Objekt geschützt werden:

```
// Attribut der Klasse
Vector vec = ...;
...

// in irgendeiner Methode
synchronized(vec)
{
  for (int i = 0; i < vec.size(); i++)
  {
    // modifiziere vec
  }
}
```

- **Verklemmungen (Deadlock)**
  Eine Verklemmung ist eine Situation, bei der Threads gegenseitig aufeinander warten und für immer blockiert bleiben.

  Coffman et al. haben gezeigt, dass bei Erfüllung der folgenden vier Bedingungen eine Verklemmung (*Deadlock*) entsteht (siehe [48]):

  1. Die Bedingung des gegenseitigen Ausschlusses: Eine Sperre ist genau von einem Thread belegt oder sie ist verfügbar.
  2. Die Belegungs- und Wartebedingung: Ein Thread, der bereits eine Sperre besitzt, darf eine weitere anfordern.
  3. Die Ununterbrechbarkeitsbedingung: Die von einem Thread belegte Sperre kann nur von ihm selbst wieder freigegeben werden.
  4. Die zyklische Wartebedingung: Es existiert eine zyklische Kette von zwei oder mehreren Threads, in der die Threads gegenseitig die Sperre eines anderen verlangen.

  Bei Java sind die ersten drei Bedingungen bereits durch die Spezifikation vorgegeben, sodass man nur auf die vierte Einfluss nehmen kann. Ein einfaches Beispiel demonstriert diese Situation:

```
public class SimpleDeadlockDemo
{
  private Object lock1 = new Object();
  private Object lock2 = new Object();

  public void methode1()
  {
    synchronized(lock1)
    {
      synchronized(lock2)
      {
        ...
      }
    }
  }


  public void methode2()
  {
    synchronized(lock2)
    {
      synchronized(lock1)
      {
        ...
      }
    }
  }
}
```

Wenn Thread 1 `methode1` aufruft, während Thread 2 nebenläufig `methode2` ausführt, kann es zu einer Verklemmung kommen (vgl. auch Abb. 4-1).

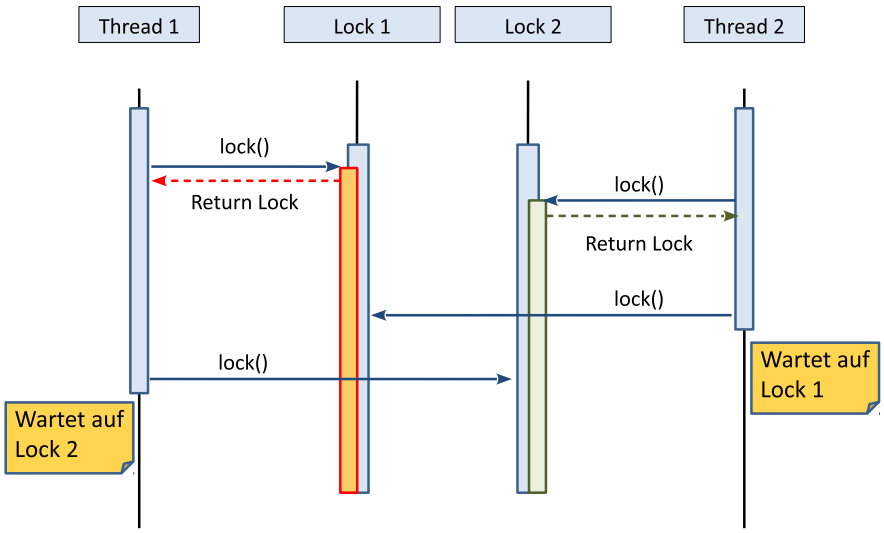

Abbildung 4-1: Deadlock durch gegenseitiges Warten

- **Aufruf von `sleep` im kritischen Bereich**
  Ein `sleep`-Aufruf in einem kritischen Bereich (innerhalb eines `synchronized`-Blocks) sollte unbedingt vermieden werden, da der schlafende Thread die Sperre nicht freigibt. Da alle darauf wartenden Threads nicht durch `interrupt` geweckt und umgeleitet werden können, kann es dazu führen, dass die ganze Anwendung stockt. Eine Sperre soll immer nur kurz belegt werden.

- **Blockierende IO im kritischen Abschnitt**
  Einige IO-Methoden sind blockierend. Das bedeutet, dass der aufrufende Thread auf ein IO-Ereignis warten muss und dabei den blockierten Zustand annimmt. Ein `interrupt` hat keine Wirkung auf sein Warteverhalten, sodass eine Verklemmung temporär auftreten kann. Im folgenden Beispiel kann kein Thread `close` aufrufen, wenn ein anderer Thread auf eine Eingabe wartet:

```
public class IOBlockMitSyncDemo
{
  private InputStream in;
  private Socket url;

  public IOBlockMitSyncDemo() throws Exception
  {
    url = new Socket("www.hs-kl.de", 80);
    in  = url.getInputStream();
  }

  public synchronized int read() throws Exception
  {
    // Situation: blockierendes Lesen,
    // die Sperre wird nicht freigegeben
    return in.read();
  }

  public synchronized void close() throws Exception
  {
    in.close();
  }
}
```

  In solchen Situationen sollte man auf andere Lösungsansätze, wie z.B. asynchrone Aufrufe, zurückgreifen.

- **Prioritätsinversion**
  Prioritätsinversion, auch Prioritätsumkehr genannt, ist ein Problem, bei dem mehrere Threads mit unterschiedlichen Prioritäten und eine Ressource, geschützt durch ein Lock-Objekt, beteiligt sind. Die Situation tritt auf, wenn ein Thread mit einer hohen Priorität auf eine Ressource zugreifen will, die von einem niedriger priorisierten Thread belegt

ist, und daher warten muss. Existiert nun ein Thread mit einer mittleren Priorität, der die fragliche Ressource nicht verwendet und eine lange Aktion durchführt, wird ihm bevorzugt die Rechenzeit zugeteilt. Der Thread mit der höchsten Priorität kommt nicht weiter, weil er indirekt durch einen mit einer niedrigen Priorität behindert wird.

Der Beinaheverlust der Pathfinder-Marssonde ist auf diesen Fehler zurückzuführen. Es gibt leider keine optimale Lösung für das Problem. Das nicht vorhersagbare Scheduling der JVM macht das Problem sogar noch schwieriger. Die beste Lösung besteht darin, dass Threads mit verschiedenen Prioritäten keine gemeinsam genutzten Ressourcen verwenden.

4.3 Zusammenfassung

Java realisiert mit dem Schlüsselwort `synchronized` ein Sperrkonzept (Ausschlussprinzip), mit dem Datenänderungen quasi atomar ausgeführt werden können. Jedes Objekt (einer Klasse) und jede Klasse besitzt hierzu eine implizite Sperre.

Ein Thread darf eine `synchronized`-Methode bzw. einen -Block nur dann betreten, wenn er die zugeordnete Sperre erworben hat. Während des Wartens auf die Freigabe kann er nicht (mit `interrupt`) unterbrochen werden. Deshalb ist es möglich, dass er für immer blockiert wird, weil die Sperre nie von dem besitzenden Thread zurückgegeben wird. Ferner können durch den Einsatz von `synchronized` auch Deadlocks entstehen.

Wie bei `volatile` werden auch für `synchronized` bestimmte Sichtbarkeiten von Variablen zugesichert. Beim Eintritt in einen `synchronized`-Block bzw. -Methode wird der Cache des lesenden bzw. aufrufenden Threads aktualisiert (*refresh*) und beim Verlassen, im Falle einer Datenänderung, in den Speicher geschrieben (*flush*). Threads, die anschließend die Sperre erwerben, erhalten die aktuellen Daten.