Nebenläufigkeit in Scala: Aktoren

Andreas de Vries

Fachhochschule Südwestfalen, Campus Hagen

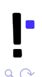
WS 2010/11

Inhalt

1. Begriffsklärungen: Threads und Nebenläufigkeit
2. Prinzipielle Probleme der Nebenläufigkeit
3. Lösungsansätze „Synchronisation“
4. Prinzipielle Probleme sequenzieller Programmiermodelle
5. Lösungsansätze „asynchrone Kommunikation“
6. Nebenläufigkeit in Scala
7. Wer war's?
8. Literatur

Übersicht

1. Begriffsklärungen: Threads und Nebenläufigkeit
2. Prinzipielle Probleme der Nebenläufigkeit
3. Lösungsansätze „Synchronisation“
4. Prinzipielle Probleme sequenzieller Programmiermodelle
5. Lösungsansätze „asynchrone Kommunikation“
6. Nebenläufigkeit in Scala
7. Wer war's?
8. Literatur



Threads, Prozesse und Prozessoren

Definition (*Prozess, Task, Thread*)

Prozess oder *Task*: ein Programm, das gestartet ist und gerade läuft.

Thread: Anweisungssequenz (*Ausführungsstrang*) innerhalb eines Prozesses.

Multitasking: Verwalten mehrerer simultaner Prozesse

Multithreading: Verwaltung mehrerer simultaner Threads

Threads, Prozesse und Prozessoren

Definition (*Prozess, Task, Thread*)

Prozess oder *Task*: ein Programm, das gestartet ist und gerade läuft.

Thread: Anweisungssequenz (*Ausführungsstrang*) innerhalb eines Prozesses.

Multitasking: Verwalten mehrerer simultaner Prozesse

Multithreading: Verwaltung mehrerer simultaner Threads

$$
\begin{aligned}
\text{Programm (Algorithmus) im RAM} &\;\widehat{=}\; \text{Kochrezept,} \\
\text{Prozessor} &\;\widehat{=}\; \text{Koch,} \\
\text{Eingabedaten} &\;\widehat{=}\; \text{Zutaten,} \\
\text{Prozess oder Task} &\;\widehat{=}\; \text{Kochen,} \\
\text{Thread} &\;\widehat{=}\; \text{Schnibbeln, Schneiden, Garen, \ldots} \\
\text{Ausgabedaten} &\;\widehat{=}\; \text{Brei.}
\end{aligned}
$$

Threads, Prozesse und Prozessoren

Definition (*Prozess, Task, Thread*)

Prozess oder *Task*: ein Programm, das gestartet ist und gerade läuft.

Thread: Anweisungssequenz (*Ausführungsstrang*) innerhalb eines Prozesses.

Multitasking: Verwalten mehrerer simultaner Prozesse

Multithreading: Verwaltung mehrerer simultaner Threads

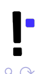
$$
\begin{aligned}
\text{Programm (Algorithmus) im RAM} &\;\widehat{=}\; \text{Kochrezept,}\\
\text{Prozessor} &\;\widehat{=}\; \text{Koch,}\\
\text{Eingabedaten} &\;\widehat{=}\; \text{Zutaten,}\\
\text{Prozess oder Task} &\;\widehat{=}\; \text{Kochen,}\\
\text{Thread} &\;\widehat{=}\; \text{Schnibbeln, Schneiden, Garen, \ldots}\\
\text{Ausgabedaten} &\;\widehat{=}\; \text{Brei.}
\end{aligned}
$$

Definition (*Nebenläufigkeit (concurrency)*)

Nebenläufigkeit: Die parallele Ausführung mehrerer Threads oder Prozesse.

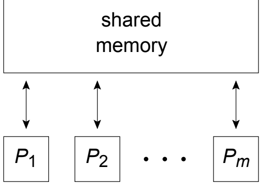
Übersicht

1. Begriffsklärungen: Threads und Nebenläufigkeit
2. Prinzipielle Probleme der Nebenläufigkeit
3. Lösungsansätze „Synchronisation“
4. Prinzipielle Probleme sequenzieller Programmiermodelle
5. Lösungsansätze „asynchrone Kommunikation“
6. Nebenläufigkeit in Scala
7. Wer war's?
8. Literatur



PRAM

- PRAM *(parallel random-access machine)*: abstraktes Maschinenmodell als Grundlage paralleler Algorithmen



- Die *m* gewöhnlichen seriell arbeitenden Prozessoren $P_1, P_2, \ldots, P_m$ teilen sich den *shared Memory*.
- Sie können gleichzeitig, also „parallel“, auf ihn lesend und schreibend zugreifen
- PRAM ist eine Verallgemeinerung des Konzepts der random-access machine (RAM) aus der Welt der sequenziellen Rechnerarchitekturen

Shared-Memory-Inkonsistenzen: Leser-Schreiber-Problem

Leser-Schreiber-Problem: Greifen mehrere Prozessoren oder Threads simultan auf gemeinsame Daten zu, so kann es zu inkonsistenten Datenzuständen kommen.

Example

Gegeben: Konto mit Kontostand 0 € und zwei Threads, die gleichzeitig 1000 € bzw. 50 € überweisen möchten. Betrachte dann folgendes Anweisungsszenario:

Zeitpunkt	Aktion	Kontostand
1.	Überweisung von Thread 1 beginnt	0 €
2.	Überweisung von Thread 2 beginnt	0 €
3.	Berechnung von Thread 1 beendet	1000 €
4.	Berechnung von Thread 2 beendet	50 €

Jeder Thread rechnete für sich korrekt, aber am Ende fehlen 1000 €!

Begrenzte Ressourcen: Deadlocks („Verklemmungen“)

Alle Threads blockieren sich gegenseitig, indem sie sich (in zyklischer Anordnung) gegenseitig notwendige Ressourcen wegnehmen: Das Gesamtsystem steht still!

©Ullenboom 2002

Übersicht

1 Begriffsklärungen: Threads und Nebenläufigkeit

2 Prinzipielle Probleme der Nebenläufigkeit

3 Lösungsansätze „Synchronisation“

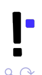
4 Prinzipielle Probleme sequenzieller Programmiermodelle

5 Lösungsansätze „asynchrone Kommunikation“

6 Nebenläufigkeit in Scala

7 Wer war's?

8 Literatur



Synchronsation durch Locks

Das Leser-Schreiber-Problem kann nur durch Sperrung des Zugriffs auf die Ressourcen gelöst werden, also:

Synchronisation durch *Locks* („Sperren“).

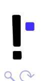
MUTEX: exklusiver Zugriff eines Threads auf eine Ressource durch Locks, d.h. Zugriff mit der Garantie, dass kein anderer Thread die Ressource liest oder verändert, solange die Sperre besteht. (⇒ Monitor, Semaphor)

Eine *Transaktion* ist eine *atomare Operation*, d.h. eine Folge von Einzeloperationen, die als eine logische Einheit betrachtet wird und entweder vollständig ausgeführt wird oder als Ganzes fehlschlägt. (Notwendig: Locks!)

Das *Transaktionsmodell* ist bei Datenbanken seit jeher praktiziert; bei nebenläufiger Programmierung: *Software Transaction Memory (SMT)* [Bra11, §11.4].

Übersicht

1. Begriffsklärungen: Threads und Nebenläufigkeit
2. Prinzipielle Probleme der Nebenläufigkeit
3. Lösungsansätze „Synchronisation“
4. Prinzipielle Probleme sequenzieller Programmiermodelle
5. Lösungsansätze „asynchrone Kommunikation“
6. Nebenläufigkeit in Scala
7. Wer war's?
8. Literatur



Nebeneffekte und Funktionen

Definition (*Nebeneffekt*)

In der prozeduralen und der objektorientierten Programmierung hat eine Methode einen *Nebeneffekt* (*side effect*) oder eine *Nebenwirkung* --- oft auch sprachlich falsch „Seiteneffekt“ ---, wenn sie durch ihre Ausführung bis auf die Speicherung des Rückgabewerts und die methodeninternen („lokalen“) Daten keine anderen Daten verändert [Bra11, S. 58], [OSV08, §§18.4, 19.7], [Pie10, S. 7, 11].

Nebeneffekte und Funktionen

Definition (*Nebeneffekt*)

In der prozeduralen und der objektorientierten Programmierung hat eine Methode einen *Nebeneffekt* (*side effect*) oder eine *Nebenwirkung* --- oft auch sprachlich falsch „Seiteneffekt“ ---, wenn sie durch ihre Ausführung bis auf die Speicherung des Rückgabewerts und die methodeninternen („lokalen“) Daten keine anderen Daten verändert [Bra11, S. 58], [OSV08, §§18.4, 19.7], [Pie10, S. 7, 11].

Definition (*Funktion*)

Eine *Funktion* f ist eine Methode, die aus einer durch ihre Schnittstellenbeschreibung *(Signatur)* eindeutig festgelegte Parameterliste $(p_1, \ldots, p_n)$ als Eingabe einen eindeutig bestimmten Wert $f(p_1, \ldots, p_n)$ berechnet [Pie10, S. 6] und keinen Nebeneffekt im obigen Sinn hat.

Der Wert einer Funktion hängt also nur von den Werten ihrer Parameter und niemals vom Zeitpunkt der Berechnung ab *(„referenzielle Transparenz“)*.

Nebeneffekte

```
1  object Nebeneffekt {
2    var sharedMemory = 0
3    def f: Int => Int = {(x) => sharedMemory * x}
4    def g: Int => Int = {(x) => sharedMemory -= x; sharedMemory}
5
6    def main(args: Array[String]) {
7      sharedMemory = 1
8      var out =   "f(1)=" + f(1) + ", g(1)=" + g(1)
9      out    += "\nf(1)=" + f(1) + ", g(1)=" + g(1)
10     javax.swing.JOptionPane.showMessageDialog(null, out, "Nebeneffekt", 2)
11   }
12 }
```

Nebeneffekte

```
1  object Nebeneffekt {
2      var sharedMemory = 0
3      def f: Int => Int = {(x) => sharedMemory * x}
4      def g: Int => Int = {(x) => sharedMemory -= x; sharedMemory}
5  
6      def main(args: Array[String]) {
7          sharedMemory = 1
8          var out =   "f(1)=" + f(1) + ", g(1)=" + g(1)
9          out     += "\nf(1)=" + f(1) + ", g(1)=" + g(1)
10         javax.swing.JOptionPane.showMessageDialog(null, out, "Nebeneffekt", 2)
11     }
12 }
```

Die Ausgabe des Programms lautet

$$\boxed{\begin{array}{ll} f(1) = 1, & g(1) = 0 \\ f(1) = 0, & g(1) = -1 \end{array}}$$

Das sind gar keine Funktionen! Genauer: f ist eine durch `sharedMemory` parametrisierte Funktion *(„Funktionenschar“)*, jedoch g verändert einen Parameter und ist daher keine Funktion, sondern … nur eine Methode.

Nebeneffekte und Funktionen

Example

Fast alle Methoden der Klasse `Math` in Java sind Funktionen, z.B. `Math.sin`, `Math.cos`, `Math.round`.

Einzige Ausnahme: `Math.random`. Sie hat zwar keine Nebeneffekte (im obigen Sinn), liefert jedoch gerade *nicht* zeitunabhängige Rückgabewerte.

Example

`System.println` und `JOptionPane.showMessageDialog` haben Nebeneffekte, denn sie verändern den Zustand des Computersystems (der Konsole bzw. des Bildschirms), obwohl sie als `void`-Methoden gemäß ihrer Schnittstellenbeschreibung keine Wirkungen zu haben vorgeben.

Nebeneffekte

Nebeneffekte sind potenziell ein Problem, wenn gemeinsame Daten von verschiedenen Algorithmen verändert werden. (Übrigens bereits in einer sequenziellen Welt, also ganz ohne Parallelität!).

Data Races

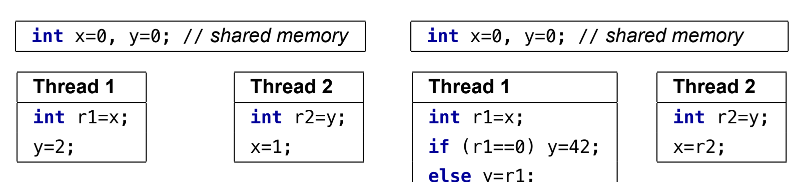
Example (Langer 2008 [LT])

```
1  import scala.actors._
2  import Actor._
3
4  object Datenwettlauf {
5     var x = 0; var y = 0; var a = 0; var b = 0;
6
7     def main(args: Array[String]) {
8        val one = actor { a = 1; x = b; } // erzeugt und startet einen Actor ...
9        Thread.sleep(100)  // ... jetzt erst kommt Indeterminismus ins Spiel ...
10       val two = actor { b = 1; y = a; }
11       System.out.println("x="+x+", y="+y+"; a="+a+", b="+b);
12    }
13 }
```

Hierbei kann durchaus $x = y = 0$ eintreten.

Sequenzielle Konsistenz

- Per se gibt Java mit seinem Speichermodell JMM *keine* Garantie der „sequenziellen Konsistenz“ [LA].
- Es kann also vorkommen, dass ein Thread eine Änderung an einer Variablen vornimmt, die ein anderer Thread aber zunächst gar nicht mitbekommt!
- Bei einem *Data race* kann in der Situation links



  durchaus der Fall `r1==1` und `r2==2` eintreten, oder sogar rechts `r1==r2==42` [AS, S. 58ff].
- Fällt bei Einkernprozessoren nicht auf, erst bei Mehrkernprozessoren.

Sequenzielle Programmierung

Wenn wir in zehn Jahren unsere Hardware als Entwickler noch auslasten wollen, wenn wir 100 Cores am Desktop-Rechner haben wollen, dann müssen wir an unseren Programmiermodellen grundsätzlich was ändern.

Klaus Alfert auf der W-JAX 09 Session

Sequenziell programmiert, könnte aber parallel ausgeführt werden:

```
for (int i = 0; i <= 256; i++) {
    x[i] = 1;
}
```

ebenso Vektoraddition:

```
for (int i = 0; i <= 256; i++) {
    z[i] = x[i] + y[i];
}
```

Sequenziell programmiert, kann aber nicht parallel ausgeführt werden:

```
x[0] = 1;
for (int i = 1; i <= 256; i++) {
    x[i] = x[i-1] + i;
}
```

Sequenzielle Programmierung

- Insbesondere sind Rekursionen streng sequenziell (Stack!).
- Der Compiler kann prinzipiell nicht selbst entscheiden (Halteproblem!?),[1] ob eine Anweisungsfolge parallel ausführbar ist.
- Der Programmierer muss die Möglichkeit haben, durch unterschiedliche Syntaxanweisungen parallele oder sequenzielle Ausführung festzulegen, z.B.:

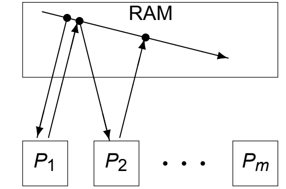
```
1   for (int i : x) {
2       i = 1;
3   }
```

- Für imperative Programmiersprachen, in denen man dem Compiler sagt, was und wie genau er was zu tun hat, würde das die Programmierung sehr schwierig machen.

[1] http://it-republik.de/jaxenter/news/JAX-TV-im-Zeichen-von-Multicore-052501.html [2010-11-16]

Amdahl'sches Gesetz

Sequenzielle Programmierung auf paralleler Prozessorarchitektur?



Amdahl'sches Gesetz:

$$\boxed{S(N) = \frac{1}{(1-p_A) + p_A/N} \leq \frac{1}{1-p_A}.} \quad (1)$$

wobei p_A der parallelisierbare Anteil der Anweisungen eines Programms ist, und $S(N)$ der Beschleunigungsfaktor (*speed-up*) bei parallelem Einsatz von N Prozessoren.

Amdahl'sches Gesetz

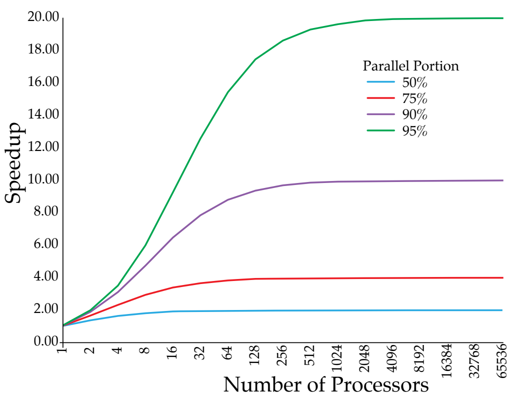
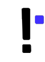
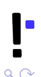


Abbildung: Das Amdahl'sche Gesetz (1): Speedup $S(N)$ abhängig von der Anzahl N der Prozessoren



Übersicht

1. Begriffsklärungen: Threads und Nebenläufigkeit
2. Prinzipielle Probleme der Nebenläufigkeit
3. Lösungsansätze „Synchronisation“
4. Prinzipielle Probleme sequenzieller Programmiermodelle
5. Lösungsansätze „asynchrone Kommunikation“
6. Nebenläufigkeit in Scala
7. Wer war's?
8. Literatur



Asynchrone Kommunikation

Definition (*Asynchrone Kommunikation*)

In Informatik und Netzwerktechnik: Kommunikation, bei der Senden und Empfangen von Nachrichten zeitlich versetzt und ohne Blockieren eines Prozesses erfolgt.

Example

E-Mail, SMS, Newsletter oder Mailing-Listen, Einträge in Diskussionsforen oder Newsgroups.

Asynchrone Kommunikation ...

- vermeidet einerseits die genannten Probleme sequenzieller Programmierung,
- impliziert andererseits jedoch den Verzicht auf *shared Memory*.

Aktorenmodell

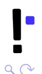
Definition (*Aktoren* und *Aktorenmodell*)

Das *Aktorenmodell* ist ein mathematisches Kommunikationsmodell nebenläufiger Systeme. Es besteht lediglich aus *Aktoren*, die sich gegenseitig Nachrichten zuschicken können. Nachrichten an einen Aktor werden in dessen Mailbox gespeichert und können von ihm nach dessen Kriterien abgearbeitet werden. Ein Aktor kann zudem weitere Aktoren erzeugen.

- Das Aktorenmodell entstand 1973 in einer Arbeit von Carl Hewitt et al. [HBS73].
- Asynchrone Kommunikation, jeder Aktor hat nur seine lokalen Daten (Cache), es gibt kein *shared Memory*.
- Eine Programmiersprache, in der das Aktorenmodell implementiert werden kann, muss daher die asynchrone Kommunikation zwischen den Aktoren ermöglichen. Hinreichende Voraussetzung: nebeneffektfreie Funktionen.

Übersicht

1 Begriffsklärungen: Threads und Nebenläufigkeit
2 Prinzipielle Probleme der Nebenläufigkeit
3 Lösungsansätze „Synchronisation“
4 Prinzipielle Probleme sequenzieller Programmiermodelle
5 Lösungsansätze „asynchrone Kommunikation“
6 Nebenläufigkeit in Scala
7 Wer war's?
8 Literatur



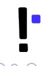
Grundsätzliches

Nebenläufigkeit in Scala basiert auf dem Aktorenmodell.

Die Actors API in Scala ist dokumentiert unter

`http://www.scala-lang.org/docu/files/actors-api/actors_api_guide.html`

Literatur: [HS11]



Syntax

```
1   import scala.actors._
2   import Actor._
3   class MyActor extends Actor {
4     ...
5     def act {
6       ... // Lebenszyklus des Aktors ...
7     }
8   }
9
10  val aktor = new MyActor
11  aktor.start
```

- Benötigt wird das Paket `scala.actors` und die Funktionen von `Actor`.
- Ein *Aktor* ist ein Objekt des Traits `Actor`,
- Der Lebenszyklus des Aktors wird in der Methode `act()` implementiert.
- Mit der Methode `start` wird ein Aktor nach Erzeugung gestartet.

Alternative Syntax

start-Methode in Konstruktor

```
1  import scala.actors._
2  import Actor._
3  class MyActor extends Actor {
4    start
5    ...
6  }
7
8  val aktor = new MyActor
```

Factory-Methode zum Erzeugen und Starten eines Aktors

```
1  import scala.actors._
2  import Actor._
3
4  val aktor = actor {
5    ... // Lebenszyklus des Aktors ...
6  }
```

Wichtige Methoden einer Actor-Klasse

Methode	Beschreibung
`start()`	Startet den Aktor
`loop`	Implementiert eine Endlosschleife
`loopWhile(`*Bedingung*`)`	Implementiert eine While-Schleife
`exit()`	Beendet den Aktor
`act()`	Implementiert den Lebenszyklus des Aktors
`receive()`	Blockiert den Aktor, bis eine Nachricht eintrifft
`react()`	Blockiert den Aktor, bis eine Nachricht eintrifft, erfordert aber weniger Overhead (Threads) als `receive`
`receiveWithin(Int)`	Wie `receive`, entblockiert jedoch nach der spezifizierten Zeit in Millisekunden
`reactWithin(Int)`	Wie `react`, entblockiert jedoch nach der spezifizierten Zeit in Millisekunden

Typischerweise sollte man zum Empfangen von Nachrichten statt `receive` oder `receiveWithin` aus Effizienzgründen möglichst `react` verwenden

Senden

Versendeart	Methode	Beschreibung
asynchron	`!`	„R `!` Case“: Nachricht an R, wartet auf keine Antwort
synchron	`!?`	„R `!?` Case“: Nachricht an R, wartet auf Antwort (`reply`) von R
synchron mit Future	`!!`	„R `!!` Case“: Nachricht an R, wartet zwar zunächst nicht auf die Antwort, aber blockiert bei späterem Abrufen des Futures

Ein Aktor ...

- empfängt Nachrichten in seiner Mailbox (Anzahl: `mailboxSize` im Object `Actor`)
- bearbeitet die nächste Nachricht in der Mailbox mit `receive` oder `react` (s.o.)

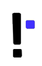
Sender und Empfänger

```
1  import scala.actors._
2  import Actor._
3
4  class Aktor(val name: String) extends Actor {
5    override def toString = name
6    def act {
7      loop {
8        react {
9          case empfänger: Aktor                =>  empfänger ! "Hallo!"
10         case ("Grüße", empfänger: Aktor)  =>  empfänger ! ("Hallo!", empfänger)
11         case "Hallo!"                        =>  println("Hallo!?")
12         case ("Hallo!", sender: Aktor)     =>  {
13           println("Hallo, "+sender+"! (Unbekannte Nachrichten: "+mailboxSize+")")
14           sender ! "Hallo, "+sender
15         }
16       }
17     }
18   }
19 }
```

Sender und Empfänger (Fortsetzung)

```
20 object SenderReceiver extends Application {
21    val sender = new Aktor("Dora"); val empfänger = new Aktor("Emil")
22    sender start; empfänger start;
23    sender ! empfänger
24    empfänger ! "Unsinn"
25    sender ! ("Grüße", empfänger)
26    System.exit(0)
27 }
```

Wer war's?



Agner Krarup Erlang (1878--1929)

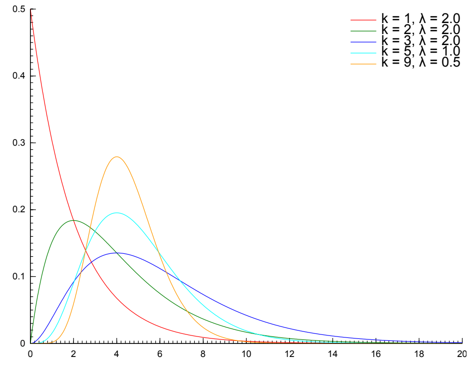
Dänischer Mathematiker und Ingenieur, forschte über Warteschlangenprobleme in der Telefonie und entwickelte die in der Nachrichtenverkehrstheorie viel verwendete Erlang-Verteilung über Blockierung und Warten in Telefonnetzen.

Eine bei Ericsson 1987 entwickelte funktionale Programmiersprache zur nebenläufigen Programmierung (und neben Java Vorbild von Scala) wurde nach ihm benannt.

Erlang-Verteilung

Wahrscheinlichkeitsdichte

$$f(x) = \begin{cases} \frac{(\lambda x)^{k-1}}{(k-1)!} \lambda \mathrm{e}^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \qquad (k \in \mathbb{N}, \lambda > 0)$$



Beschreibt die Verteilung der Zeitspanne zwischen k Ereignissen eines Poisson-Prozesses

z.B. Ankunft von Kunden, oder Lebensdauern in der Qualitätssicherung

Diskussion

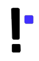
Braun, Oliver:
Scala. Objektfunktionale Programmierung.
München : Carl Hanser Verlag, 2011

Hewitt, Carl ; Bishop, Peter ; Steiger, Richard:
`A Universal Modular Actor Formalism for Artificial Intelligence'.
In: *International Joint Conference on Artificial Intelligence* (1973), S. 235--245

Haller, Philipp ; Sommers, Frank:
Actors in Scala.
Mountain View : Artima Press, 2011

Odersky, Martin ; Spoon, Lex ; Venners, Bill:
Programming in Scala.
Mountain View : Artima Press, 2008

Piepmeyer, Lothar:
Grundkurs Funktionale Programmierung mit Scala.
München Wien : Carl Hanser Verlag, 2010. --
www.grundkurs-funktionale-programmierung.de



Netzquellen

[AS] http://www.inf.ed.ac.uk/teaching/courses/apl/2008-2009/slides/apl19.pdf -- D. Aspinall: *Cautionary Tales in Concurrency*, Lecture Notes *Advances in Programming Languages*, University of Edinburgh (2009), [2013-10-27]

[JLS] http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html -- Java Language Specification, Third Edition, Chapter 17: Threads and Locks [2013-10-27]

[LA] http://www.angelikalanger.com/Articles/EffectiveJava/38.JMM-Overview/38.JMM-Overview.html -- Angelika Langer & Klaus Kreft: *Java Memory Model: Überblick* [2013-10-27]

[LT] http://it-republik.de/jaxenter/news/JAX-TV-Java-Programmierung-im-Multicore-Zeitalter-047030.html -- Angelika Langer: `Java Programming in a Multicore World', Vortrag auf der jax 2008 [2010-11-17], siehe auch http://www.youtube.com/watch?v=gkIgo892N_w [2013-10-27]

[SA] http://www.scala-lang.org/docu/files/actors-api/actors_api_guide.html -- Scala Actor API [2013-10-27]