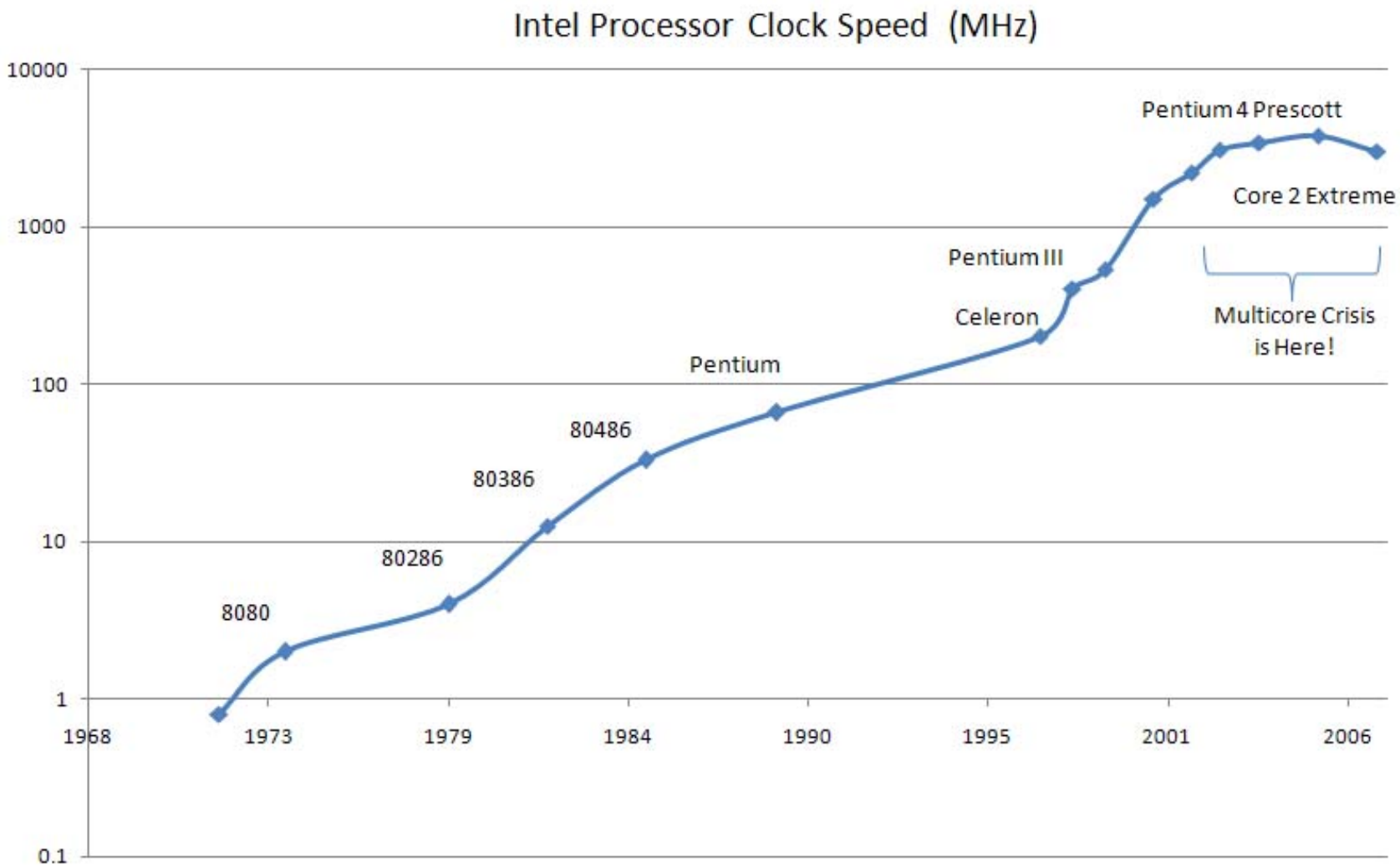
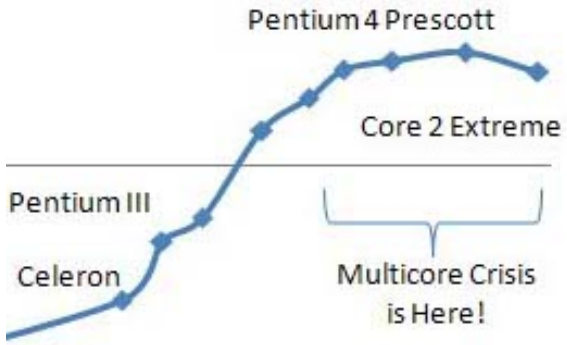
Softwarearchitekturen für die Entwicklung verteilter Anwendungen

Dominik Charousset

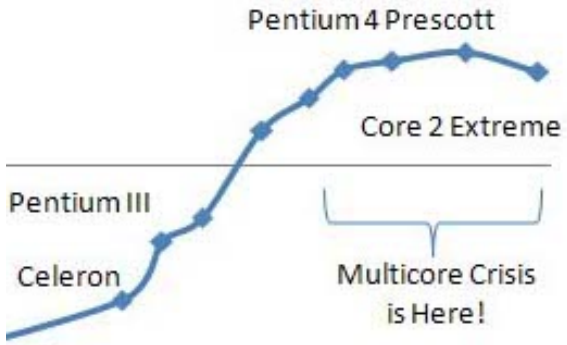
SoSe 09

Gliederung


Intel Processor Clock Speed (MHz)
10000
1000
100
10
1
0.1
1968
1973
1979
1984
1990
1995
2001
2006
8080
80286
80386
80486
Pentium
Celeron
Pentium III
Pentium 4 Prescott
Core 2 Extreme
Multicore Crisis
is Here!



- *Clock Speed* der Prozessoren steigt nicht mehr wesentlich
- Aber: Zahl der Prozessor**kerne** steigt stetig
- Nicht-parallele Software profitiert nicht von neuer Hardware



- *Clock Speed* der Prozessoren steigt nicht mehr wesentlich
- Aber: Zahl der Prozessor**kerne** steigt stetig
- Nicht-parallele Software profitiert nicht von neuer Hardware

"Software has to double the amount of parallelism that it can support every two years." – Shekhar Y. Borkar (Intel)

In multithreaded Anwendungen muss ...

1. jedes Objekt (im *Shared Memory*) ***thread safe*** sein
2. jede Operationen auf einem *stateful object* synchronisiert werden

In multithreaded Anwendungen muss ...

1. jedes Objekt (im *Shared Memory*) ***thread safe*** sein
2. jede Operationen auf einem *stateful object* synchronisiert werden

Die Verantwortung liegt beim Entwickler; Fehler führen u.A. zu:

- Race Conditions
- Deadlocks
- Lifelocks

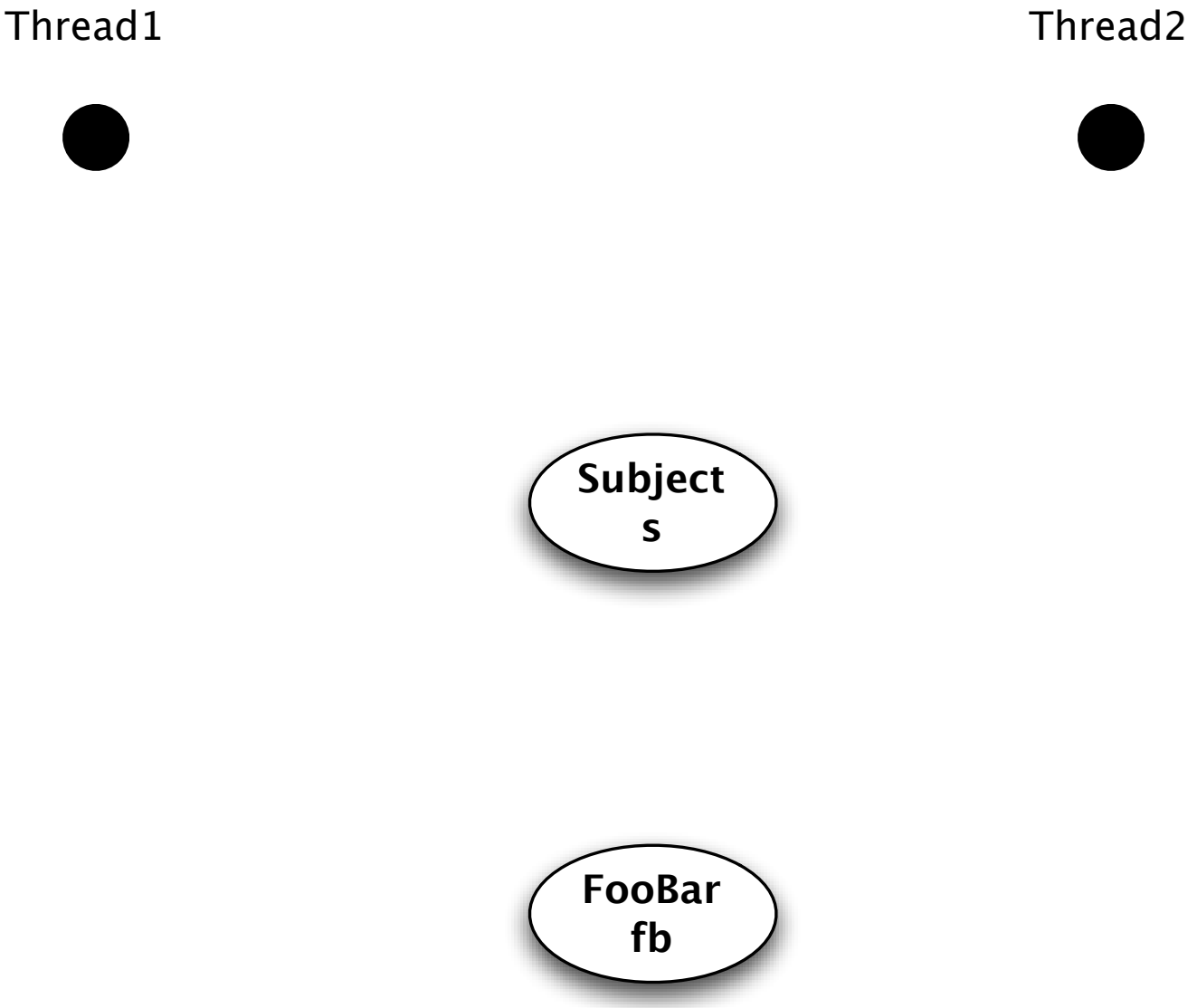
- Multithreaded Anwendungen mit Shared Memory sind:
  - schwer zu schreiben und zu durchschauen
  - selbst durch systematisches testen nicht verifizierbar
- Selbst scheinbar sichere Codeteile können zu Deadlocks oder Speicherverletzungen führen

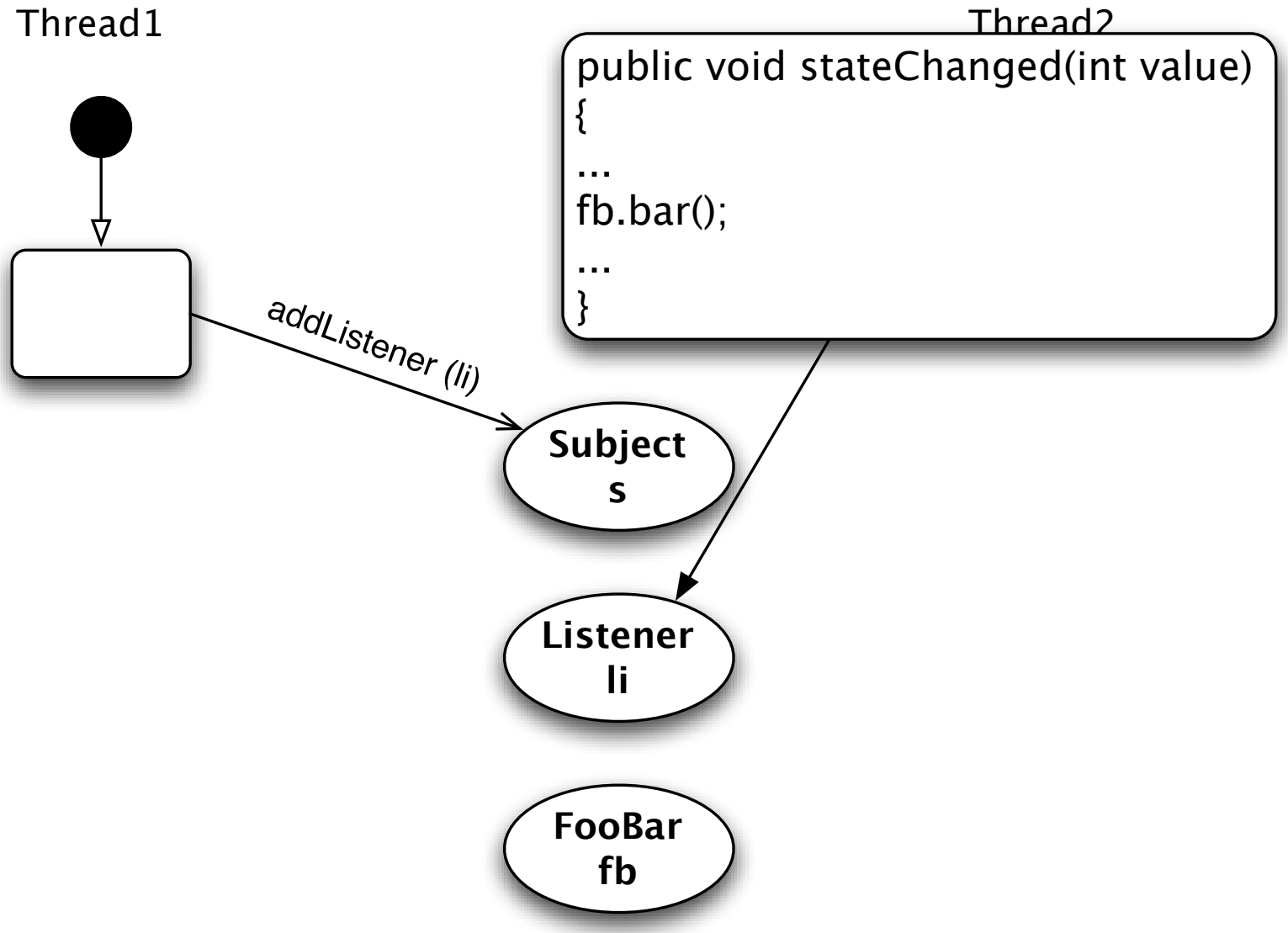
- Multithreaded Anwendungen mit Shared Memory sind:
  - schwer zu schreiben und zu durchschauen
  - selbst durch systematisches testen nicht verifizierbar
- Selbst scheinbar sichere Codeteile können zu Deadlocks oder Speicherverletzungen führen

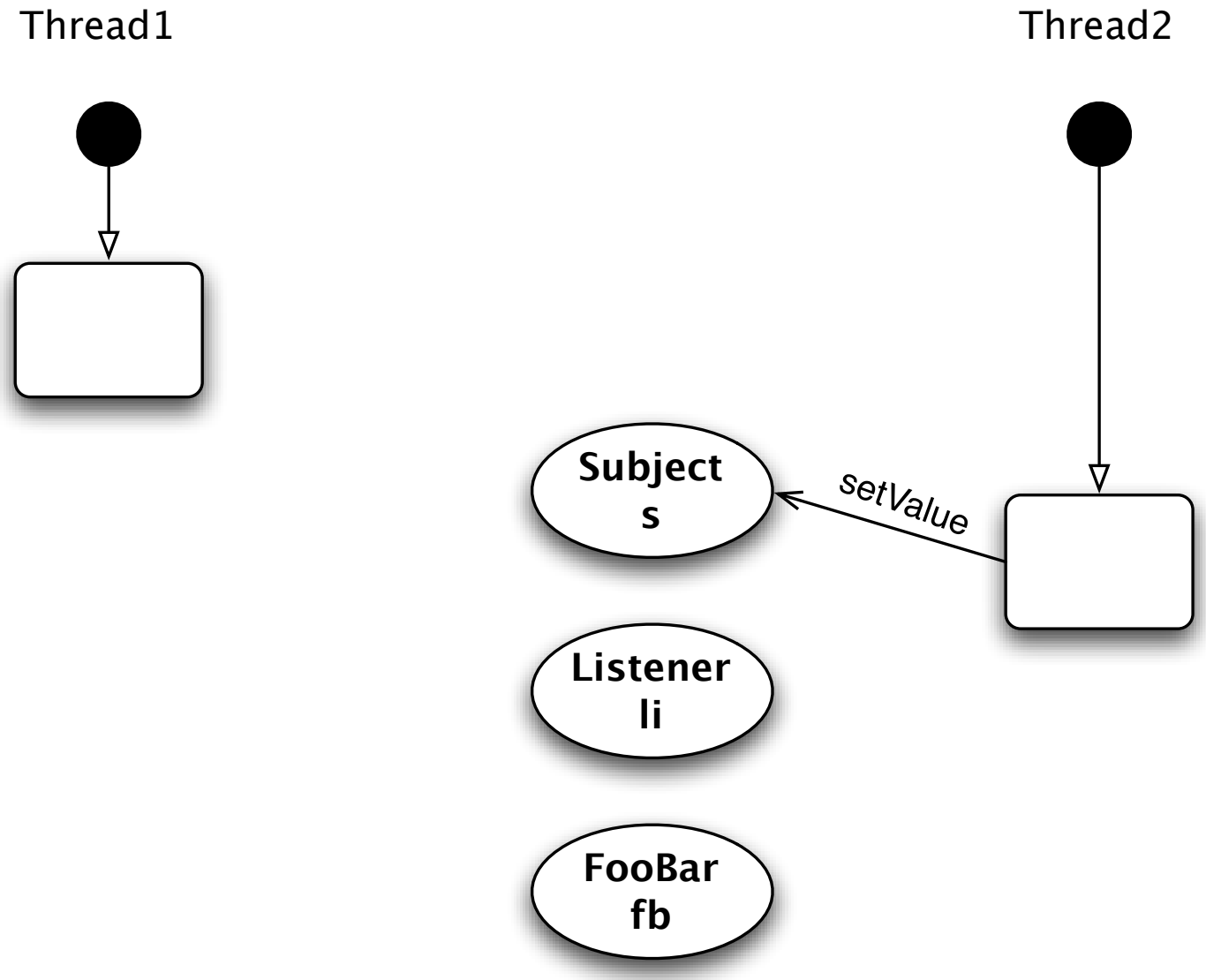
“Mutable stateful objects are the new spaghetti code” – Rich Hickey

```
class Subject {
    private int value; private List<Listener> listeners = ...;
    public interface Listener {
        public void stateChanged(int newValue);
    }
    public synchronized void addListener(Listener listener) {
        listeners.add(listener);
    }
    public synchronized void setValue(int newValue) {
        value = newValue;
        for (Listener l : listeners) {
            l.stateChanged(newValue);
        }
}
```

```
class FooBar {
    private Subject s;
    public FooBar(Subject s) { this.s= s; }
    public synchronized void bar() { /* unwichtig */ }
    public synchronized void foo() {
        ...
        s.addListener(...);
        ...
    }
}
```

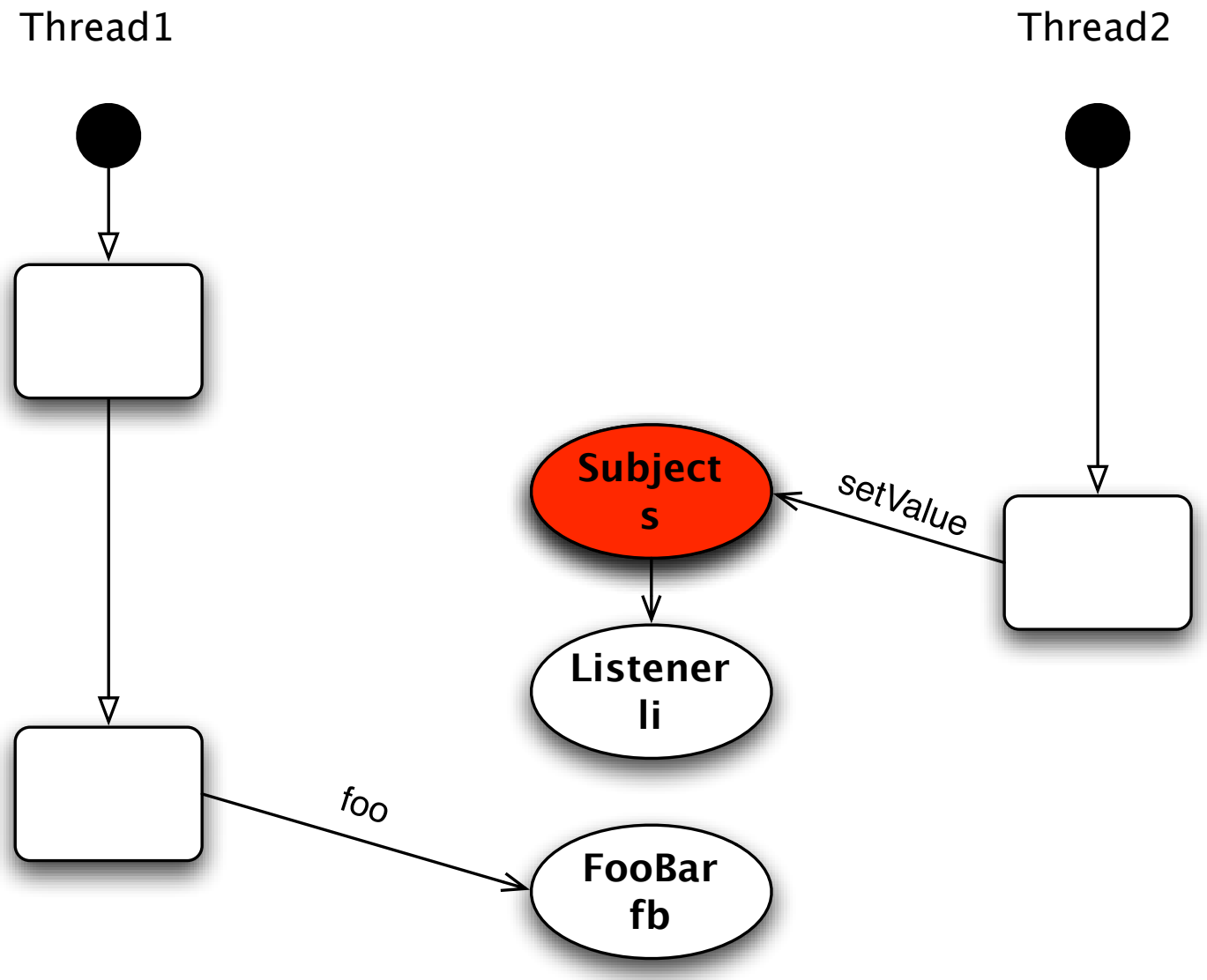

Thread1
Thread2
Subject
s
FooBar
fb


Thread1
Thread2
public void stateChanged(int value)
{
...
fb.bar();
...
}
addListener (li)
Subject
s
Listener
li
FooBar
fb


Thread1
Thread2
Subject
s
setValue
Listener
li
FooBar
fb

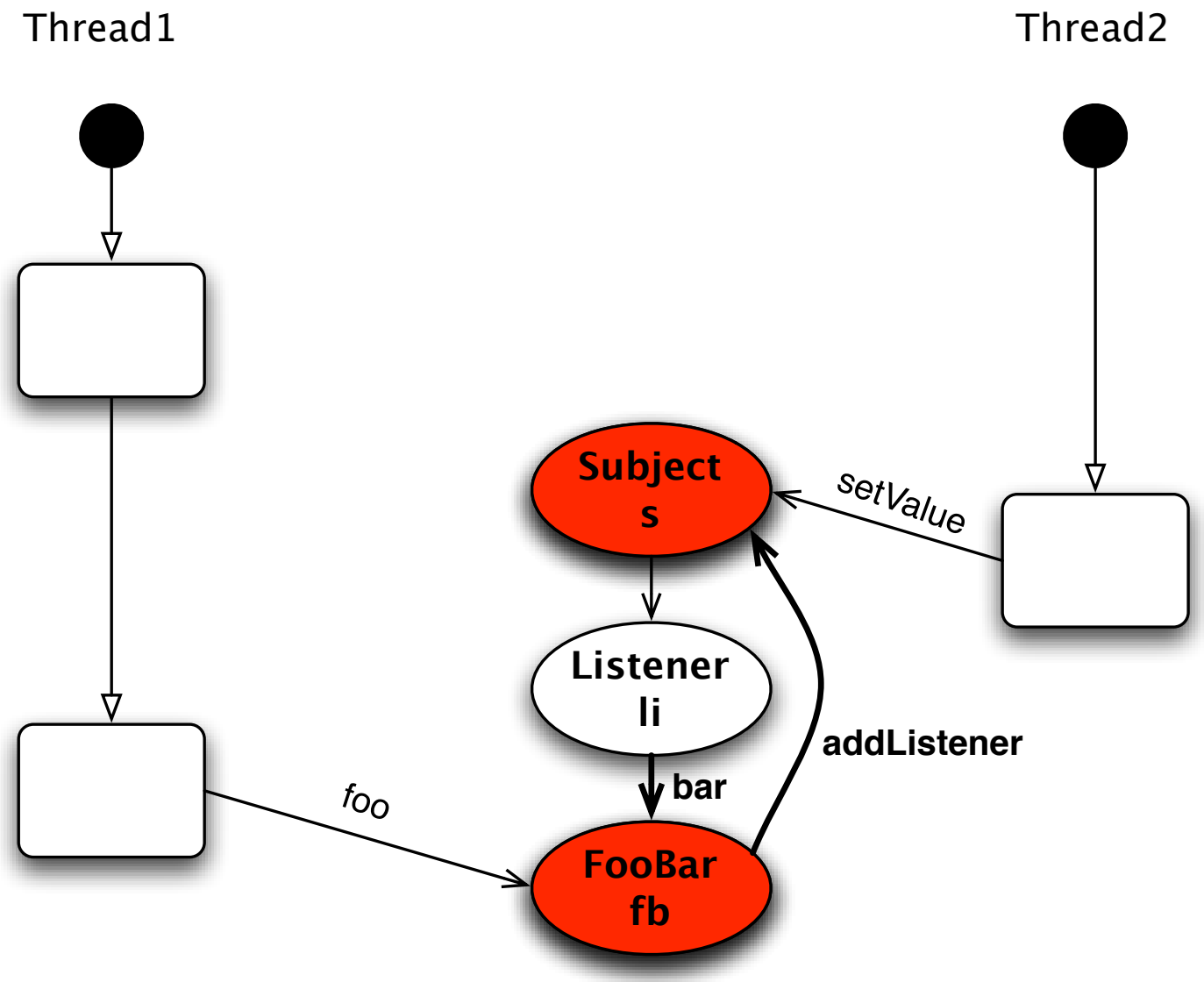
Einführung

Multithreading & Shared Memory – Beispiel 1



Einführung

Multithreading & Shared Memory – Beispiel 1



Quellcode:

```
Singleton* Singleton::instance() {
    // 1st test
    if (pInstance == 0) {
        Lock lock;
        // 2nd test
        if (pInstance == 0) {
            pInstance = new Singleton;
        }
    }
    return pInstance;
}
```

Übernommen aus "*C++ and the Perils of Double-Checked Locking*" (Meyers & Alexandrescu, 2004)

Quellcode:

```
Singleton* Singleton::instance() {
    // 1st test
    if (pInstance == 0) {
        Lock lock;
        // 2nd test
        if (pInstance == 0) {
            pInstance = new Singleton;
        }
    }
    return pInstance;
}
```

Problem:

pInstance = **new** Singleton:

1. Speicher allokieren
2. Konstruktor ausführen
3. Adresse *pInstance* zuweisen

Übernommen aus "*C++ and the Perils of Double-Checked Locking*" (Meyers & Alexandrescu, 2004)

Quellcode:

```
Singleton* Singleton::instance() {
    // 1st test
    if (pInstance == 0) {
        Lock lock;
        // 2nd test
        if (pInstance == 0) {
            pInstance = new Singleton;
        }
    }
    return pInstance;
}
```

Problem:

pInstance = **new** Singleton:

1. Speicher allokieren
2. Konstruktor ausführen
3. Adresse *pInstance* zuweisen

Ein Fehler tritt auf, wenn:

- Compiler 3. vor 2. setzt
- Ein Thread pInstance dereferenziert, obwohl noch nicht initialisiert

Übernommen aus "*C++ and the Perils of Double-Checked Locking*" (Meyers & Alexandrescu, 2004)

Related Works

Das Aktormodell

Aktoren ...

- arbeiten parallel
- teilen sich **keinen** gemeinsamen Speicher
- kommunizieren über asynchrone Nachrichten
- können zur Laufzeit weitere Aktoren erschaffen

Die Referenzimplementierung eines Aktormodells ist **Erlang**:

- Namensdienst für Aktoren
- **Kein** Shared Memory
- *Supervision Tree*
- *Pattern Matching* auf einkommende Nachrichten
- Aktoren können auf beliebigen Nodes gestartet werden
  - $\Rightarrow$ “global” verteilte Anwendung

Related Works

Clojure – (Software) Transitional Memory (STM)

- Zugriffe in den Speicher sind Transaktionen
- Schreib- und Leseoperationen sind atomar und “*all or nothing*”
- Keine Race Conditions möglich
- Keine Locks!
- Leser blockieren keine Schreiber & vice versa
- Bei Clojure fest in Sprache integriert
- Der Clojure STM ist eine *snapshot MVCC*
  (MVCC = Multiversion Concurrency Control)

Related Works

Clojure – Agenten

- Agenten arbeiten parallel (*reactive*)
- Haben einen *abfragbaren* Zustand
- Nachrichten sind Funktionen, die der Agent ausführen soll

  $\Rightarrow$ `(send agent fn args*)`
- Agenten sind (asynchrone) Funktionen ihres Zustandes

  $\Rightarrow$ `state = fn(state)`

Related Works

Join-Calculus (am Beispiel von JoCaml)

```
1: def fruit(f) & cake(c) = print_endline (f^“ ”^c) ; 0
   val fruit : string Join.chan = <abstr>
   val cake : string Join.chan = <abstr>

2: spawn fruit(“apple”) & cake(“pie”)

3: spawn fruit “apple” & fruit “lime” & cake “pie” & cake “torte”
```

- Join-Calculus stammt aus der Familie der π-Calculi
- Mathematisches Modell für Prozesse

Related Works

Join-Calculus (am Beispiel von JoCaml)

```
1: def fruit(f) & cake(c) = print_endline (f^“ ”^c) ; 0
   val fruit : string Join.chan = <abstr>
   val cake : string Join.chan = <abstr>

2: spawn fruit(“apple”) & cake(“pie”)

3: spawn fruit “apple” & fruit “lime” & cake “pie” & cake “torte”
```

- Join-Calculus stammt aus der Familie der π-Calculi
- Mathematisches Modell für Prozesse
- Im Beispiel:
  1. Definition der (async) Channel und ihres *gemeinsamen* verhaltens
  2. Gibt “apple pie” aus
  3. Gibt “apple pie” & “lime torte” oder “apple torte” & “lime pie” aus

Related Works

MapReduce (© Google)

- Gedacht für Operationen auf großen Datenmengen in Clustern
- Framework verteilt automatisch Aufgaben an die Worker im Cluster
- Verfahren besteht aus zwei Funktionen:

map erzeugt Zwischenresultate als key/value pairs

reduce reduziert Zwischenresultate bis hin zum “Endergebnis”

Code aus “*MapReduce: simplified data processing on large clusters*” (Dean & Ghemawat, 2008)

Related Works

MapReduce (© Google)

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

- Gedacht für Operationen auf großen Datenmengen in Clustern
- Framework verteilt automatisch Aufgaben an die Worker im Cluster
- Verfahren besteht aus zwei Funktionen:

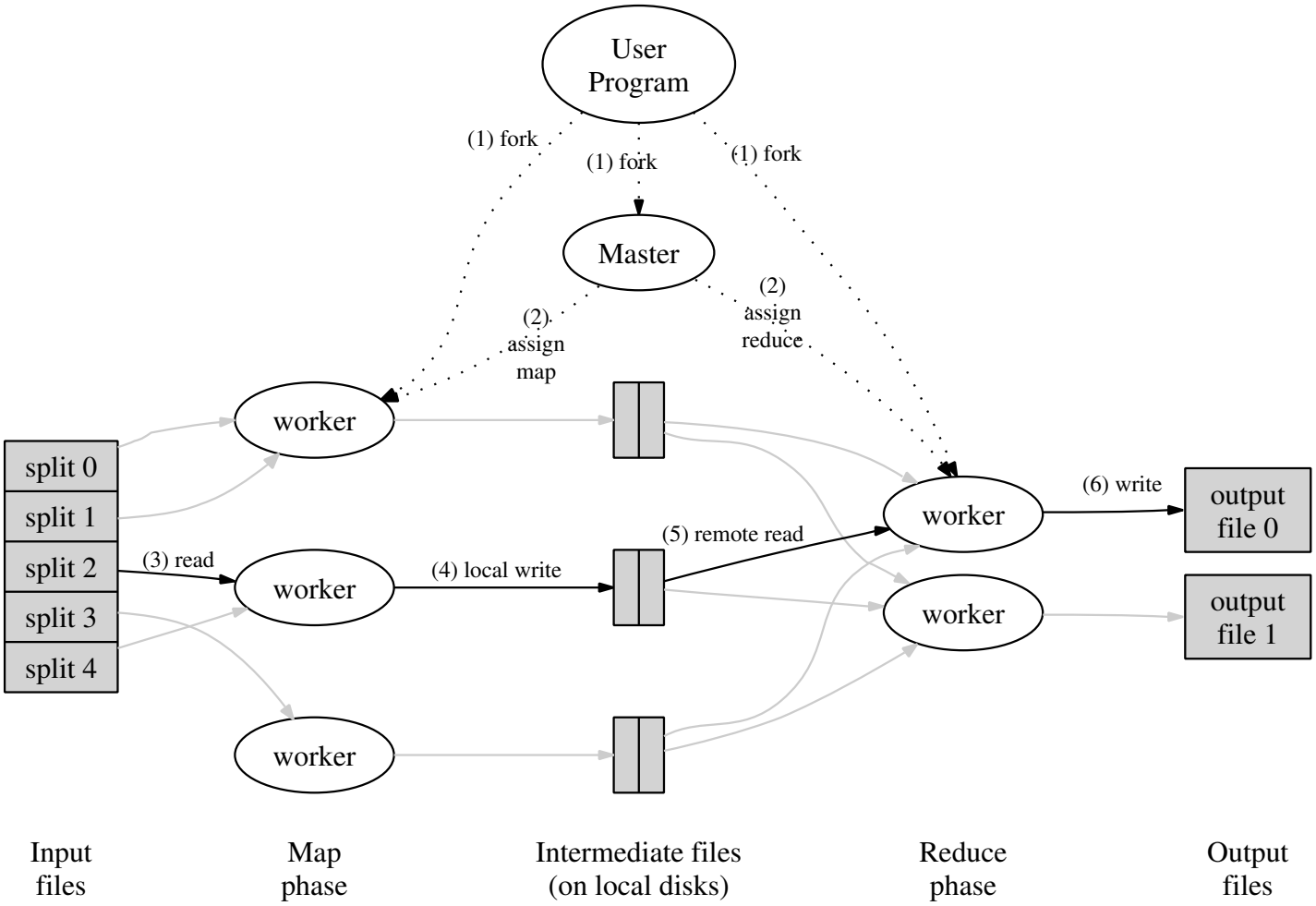
map erzeugt Zwischenresultate als key/value pairs

reduce reduziert Zwischenresultate bis hin zum “Endergebnis”

Code aus “*MapReduce: simplified data processing on large clusters*” (Dean & Ghemawat, 2008)

Related Works

MapReduce (© Google)



Graphik aus "*MapReduce: simplified data processing on large clusters*" (Dean & Ghemawat, 2008)

Eigene Arbeit

Aktuelles Masterprojekt im SoSe 09

- Verteilte Anwendungen
- Schwerpunkt auf Aktoren
- Implementierung eines (Erlang-like) Aktormodells als Bibliothek in C++

Related Projects

Scala

- Hybridsprache (Funktional + OO)
- Läuft auf der JVM und ist voll kompatibel zu Java
  - $\Rightarrow$ wie Clojure (aber Clojure ist rein funktional)
- Implementiert ein Aktormodell (seit Version 2.1.7)
  - Stark an Erlang angelehnt
  - Und: Scala hat ebenfalls *Pattern Matching*
- Prinzipiell Shared Memory Zugriffe zwischen Aktoren möglich
  - $\Rightarrow$ aber durch den funkt. Stil von Scala verpöhnt
  - $\Rightarrow$ Scala arbeitet in erster Linie mit *immutable* Objekten

Related Projects

Kilim

- Framework für Java
- Bytecode-Postcompiler
  - wertet Kilim-eigene Annotations aus
  - statische Analyse der Aktoren auf Isolation
  - modifiziert Nachrichtentypen
- Keine Shared Memory Zugriffe zwischen Aktoren erlaubt
- Kein Mapping von Aktoren auf native Threads (*CPS transformation*)
- Keine beliebigen Java-Objekte als Nachrichten
- Aktuelle Version: 0.6

Related Projects

Retlang

- Framework für C#
- Aktoren sind sog. “Fiber”
- Kommunikation über getypte Channel
  - Fiber werden mit “Callback” am Channel angemeldet
  - Callback wird im Kontext des Fibers ausgeführt
- Aktuelle Version: 0.4.2

Zusammenfassung

- Verteilte Anwendungen brauchen Infrastruktur
- Es gibt nicht **die** Lösung / Architektur für verteilte Anwendungen
- Jedes Modell hat Stärken und Schwächen

Zusammenfassung

- Verteilte Anwendungen brauchen Infrastruktur
- Es gibt nicht **die** Lösung / Architektur für verteilte Anwendungen
- Jedes Modell hat Stärken und Schwächen
- Das Aktormodell wurde als Projektthema gewählt, da es ...
  - einfach und intuitiv zu verstehen ist
  - Verteilung über Netzwerke *sehr* einfach macht
  - (fast) beliebig skaliert

Zusammenfassung

- Verteilte Anwendungen brauchen Infrastruktur
- Es gibt nicht **die** Lösung / Architektur für verteilte Anwendungen
- Jedes Modell hat Stärken und Schwächen
- Das Aktormodell wurde als Projektthema gewählt, da es ...
  - einfach und intuitiv zu verstehen ist
  - Verteilung über Netzwerke *sehr* einfach macht
  - (fast) beliebig skaliert

“Use the right tool for the right job!”

Quellen

- ARMSTRONG, Joe: *Making Reliable Distributed Systems in the Presence of Software Errors*. 2003. – URL `http://erlang.org/download/armstrong_thesis_2003.pdf`

- ARMSTRONG, Joe: A history of Erlang. In: *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. New York, NY, USA : ACM, 2007, S. 6–1–6–26. – ISBN 978-1-59593-766-X

Quellen

- DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: simplified data processing on large clusters. In: *Commun. ACM* 51 (2008), Nr. 1, S. 107–113. – ISSN 0001-0782

- DICE, Dave ; LEV, Yossi ; MOIR, Mark ; NUSSBAUM, Daniel: Early experience with a commercial hardware transactional memory implementation. In: *SIGPLAN Not.* 44 (2009), Nr. 3, S. 157–168. – ISSN 0362-1340

Quellen

- HALLER, Philipp ; ODERSKY, Martin: Scala Actors: Unifying thread-based and event-based programming. In: *Theor. Comput. Sci.* 410 (2009), Nr. 2-3, S. 202–220. – ISSN 0304-3975

- MEYERS, Scott ; ALEXANDRESCU, Andrei: C++ and the Perils of Double-Checked Locking. (2004)

Vielen Dank für Ihre Aufmerksamkeit!

Fragen?