Performance and Scalability (11)

Christian Schürmann, Simon Moissl

27.06.2008

1. Thinking about performance
2. Amdahl's Law
3. Costs introduced by threads
4. Reducing lock contention
5. Outlook & Questions

Über Performanz

Pro und Contra Threads...

+

- höhere Performanz
- höhere Ressourcenauslastung
- bessere Ansprechsensibilität (responsiveness)

–

- Overhead für locking, signalling, memory synchronization
- (im Allgemeinen) mehr Kontextwechsel
- Erstellung (thread creation overhead)

Threads nur benutzten wenn Performanzgewinn die Kosten übersteigt

Über Performanz

Pro und Contra Threads...

- höhere Performanz
- höhere Ressourcenauslastung
- bessere Ansprechsensibilität (responsiveness)

–

- Overhead für locking, signalling, memory synchronization
- (im Allgemeinen) mehr Kontextwechsel
- Erstellung (thread creation overhead)

Threads nur benutzten wenn Performanzgewinn die Kosten übersteigt

Über Performanz

Pro und Contra Threads...

+

- höhere Performanz
- höhere Ressourcenauslastung
- bessere Ansprechsensibilität (responsiveness)

-

- Overhead für locking, signalling, memory synchronization
- (im Allgemeinen) mehr Kontextwechsel
- Erstellung (thread creation overhead)

Threads nur benutzten wenn Performanzgewinn die Kosten übersteigt

Über Performanz

Pro und Contra Threads...

+

- höhere Performanz
- höhere Ressourcenauslastung
- bessere Ansprechsensibilität (responsiveness)

-

- Overhead für locking, signalling, memory synchronization
- (im Allgemeinen) mehr Kontextwechsel
- Erstellung (thread creation overhead)

Threads nur benutzten wenn Performanzgewinn die Kosten übersteigt

Performanz versus Skalierbarkeit

Definition

Skalierbarkeit beschreibt die Fähigkeit den Durchsatz oder die Kapazität zu verbessern, indem man zusätzliche Ressourcen hinzufügt

Optimierung bzgl. Performanz

- Gleiche Arbeit in weniger Zeit
- z.B. $O(n^2) \leftrightarrow O(n \log n)$

Optimierung bzgl. Skalierbarkeit

- Mehr Arbeit mit mehr Ressourcen
- z.B. durch mehr Prozessoren

Performanz versus Skalierbarkeit

Definition

Skalierbarkeit beschreibt die Fähigkeit den Durchsatz oder die Kapazität zu verbessern, indem man zusätzliche Ressourcen hinzufügt

Optimierung bzgl. Performanz

- Gleiche Arbeit in weniger Zeit
- z.B. $O(n^2) \leftrightarrow O(n \log n)$

Optimierung bzgl. Skalierbarkeit

- Mehr Arbeit mit mehr Ressourcen
- z.B. durch mehr Prozessoren

Performanz versus Skalierbarkeit

Definition

Skalierbarkeit beschreibt die Fähigkeit den Durchsatz oder die Kapazität zu verbessern, indem man zusätzliche Ressourcen hinzufügt

Optimierung bzgl. Performanz

- Gleiche Arbeit in weniger Zeit
- z.B. $O(n^2) \leftrightarrow O(n \log n)$

Optimierung bzgl. Skalierbarkeit

- Mehr Arbeit mit mehr Ressourcen
- z.B. durch mehr Prozessoren

Einschätzen von Performanz-Kompromissen

Voreiliges Optimieren vermeiden!

Es gilt: Erst lauffähigen Code produzieren, dann optimieren...
... (wenn überhaupt notwendig)

Um zu entscheiden, welcher Code schneller ist, fragt man:

- In welchem Kontext bin ich schneller (heavy vs. light load)
- Wie oft tritt mein Kontext ein
- Wird Code in anderen Kontexten (andere Bedingungen) verwendet
- Versteckte Kosten: Schlechtere Wartbarkeit etc.

Measure, don't guess!

Measure... measure again... and again... and again...

Einschätzen von Performanz-Kompromissen

Voreiliges Optimieren vermeiden!

Es gilt: Erst lauffähigen Code produzieren, dann optimieren...
... (wenn überhaupt notwendig)

Um zu entscheiden, welcher Code schneller ist, fragt man:

- In welchem Kontext bin ich schneller (heavy vs. light load)
- Wie oft tritt mein Kontext ein
- Wird Code in anderen Kontexten (andere Bedingungen) verwendet
- Versteckte Kosten: Schlechtere Wartbarkeit etc.

Measure, don't guess!

Measure... measure again... and again... and again...

Einschätzen von Performanz-Kompromissen

Voreiliges Optimieren vermeiden!

Es gilt: Erst lauffähigen Code produzieren, dann optimieren...
... (wenn überhaupt notwendig)

Um zu entscheiden, welcher Code schneller ist, fragt man:

- In welchem Kontext bin ich schneller (heavy vs. light load)
- Wie oft tritt mein Kontext ein
- Wird Code in anderen Kontexten (andere Bedingungen) verwendet
- Versteckte Kosten: Schlechtere Wartbarkeit etc.

Measure, don't guess!

Measure... measure again... and again... and again...

Einschätzen von Performanz-Kompromissen

Voreiliges Optimieren vermeiden!

Es gilt: Erst lauffähigen Code produzieren, dann optimieren...
... (wenn überhaupt notwendig)

Um zu entscheiden, welcher Code schneller ist, fragt man:

- In welchem Kontext bin ich schneller (heavy vs. light load)
- Wie oft tritt mein Kontext ein
- Wird Code in anderen Kontexten (andere Bedingungen) verwendet
- Versteckte Kosten: Schlechtere Wartbarkeit etc.

Measure, don't guess!

Measure... measure again... and again... and again...

Einschätzen von Performanz-Kompromissen

Voreiliges Optimieren vermeiden!

Es gilt: Erst lauffähigen Code produzieren, dann optimieren...
... (wenn überhaupt notwendig)

Um zu entscheiden, welcher Code schneller ist, fragt man:

- In welchem Kontext bin ich schneller (heavy vs. light load)
- Wie oft tritt mein Kontext ein
- Wird Code in anderen Kontexten (andere Bedingungen) verwendet
- Versteckte Kosten: Schlechtere Wartbarkeit etc.

Measure, don't guess!

Measure... measure again... and again... and again...

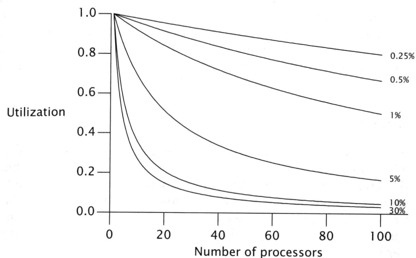
Amdahlsches Gesetz

Einige Aufgaben können parallelisiert werden (harvest),
andere nicht (grow)

Amdahlsches Gesetz

$$S \leq \left(F + \frac{1-F}{N}\right)^{-1}$$

F = Anteil an sequentiellem Code
N = Anzahl der Prozessoren
S = Geschwindigkeitsgewinn durch Parallelisierung



Die meisten Anwendungen haben sowohl sequentiellen als auch parallelen Codeanteil

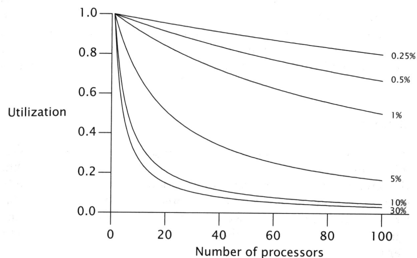
Amdahlsches Gesetz

Einige Aufgaben können parallelisiert werden (harvest),
andere nicht (grow)

Amdahlsches Gesetz

$$S \leq \left(F + \frac{1-F}{N}\right)^{-1}$$

F = Anteil an sequentiellem Code
N = Anzahl der Prozessoren
S = Geschwindigkeitsgewinn durch Parallelisierung



Die meisten Anwendungen haben sowohl sequentiellen als auch parallelen Codeanteil

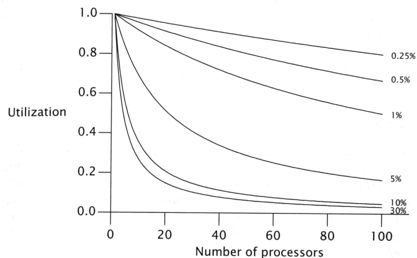
Amdahlsches Gesetz

Einige Aufgaben können parallelisiert werden (harvest),
andere nicht (grow)

Amdahlsches Gesetz

$$S \leq \left(F + \frac{1-F}{N}\right)^{-1}$$

F = Anteil an sequentiellem Code
N = Anzahl der Prozessoren
S = Geschwindigkeitsgewinn durch Parallelisierung



Die meisten Anwendungen haben sowohl sequentiellen als auch parallelen Codeanteil

Zwischenfrage

Frage:

Wenn der parallele Codeanteil in den Threads steckt, wo ist dann der sequentielle Codeanteil?

Antwort:

Die Threads holen sich Ihre Aufgaben z.B. aus einer Liste
I/O auf der Liste (i.d.R.) ist sequentiell

Zwischenfrage

Frage:

Wenn der parallele Codeanteil in den Threads steckt, wo ist dann der sequentielle Codeanteil?

Antwort:

Die Threads holen sich Ihre Aufgaben z.B. aus einer Liste
I/O auf der Liste (i.d.R.) ist sequentiell

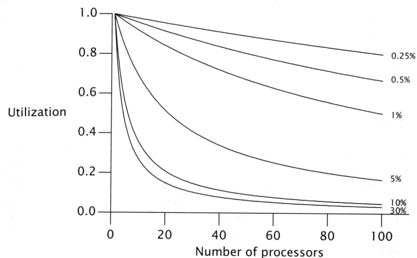
Amdahlsches Gesetz

Einige Aufgaben können parallelisiert werden (harvest),
andere nicht (grow)

Amdahlsches Gesetz

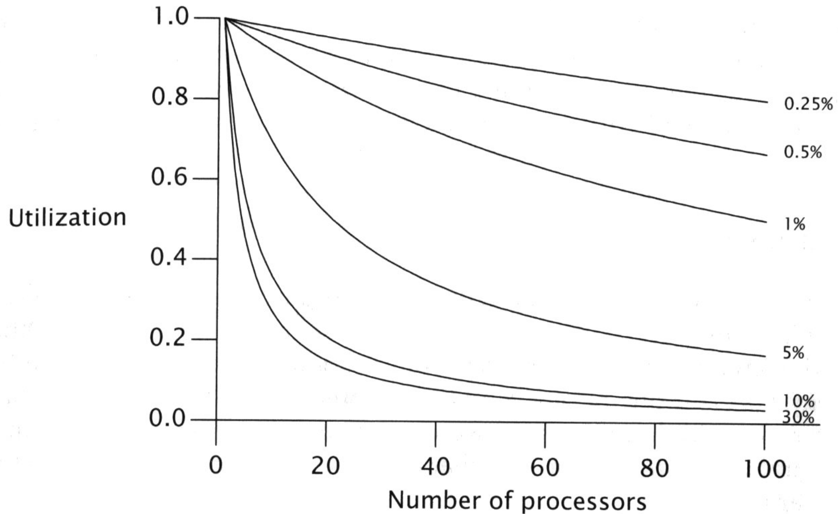
$$S \leq \left(F + \frac{1-F}{N}\right)^{-1}$$

F = Anteil an sequentiellem Code
N = Anzahl der Prozessoren
S = Geschwindigkeitsgewinn durch Parallelisierung



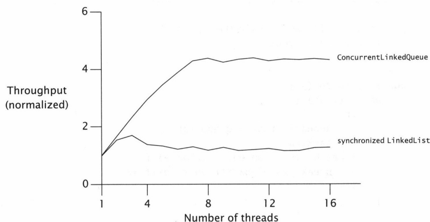
Alle Anwendungen haben sowohl sequentiellen als auch parallelen Codeanteil

Amdahlsches Gesetz



Beispiel: Listenimplementierungen

Verschiedene Listen haben unterschiedlichen sequentiellen Codeanteil

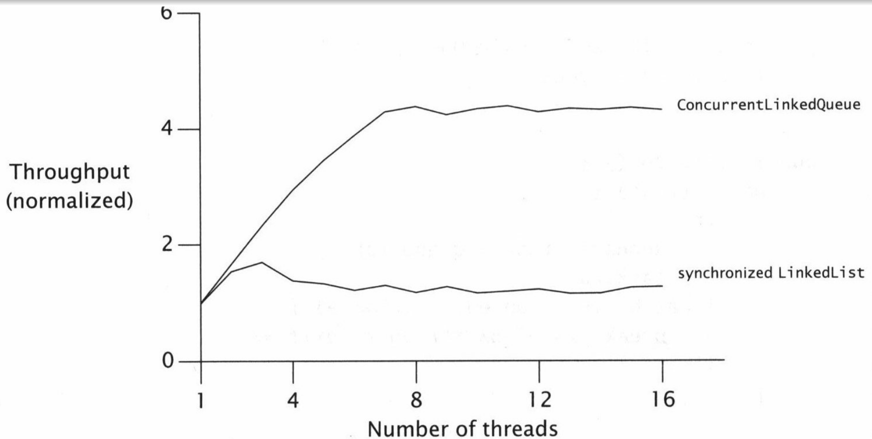


Amdahlsches Gesetz

$$S = (F + O(P) + \frac{1-F}{N})^{-1}$$

F = Anteil an sequentiellem Code
N = Anzahl der Prozessoren
S = Geschwindigkeitsgewinn durch Parallelisierung
O(P) = Kommunikation(soverhead) zwischen Prozessoren (Nodes)

Beispiel: Listenimplementierungen



Kontextwechsel

Definition

Kontextwechsel nennt man den Vorgang, bei dem die Bearbeitung des aktuellen Threads unterbrochen und zu einem anderen Thread gewechselt wird.
Dabei wird der Kontext (im Wesentlichen die Prozessor-Register) des aktuellen Threads gesichert und der Kontext des neuen restauriert.

Kontextwechsel...

- ... sind nicht kostenfrei
- ... brauchen selbst CPU-Zeit
- ... und verbrauchen somit Ressourcen

Die tatsächlichen Kosten sind plattformabhängig.

Daumenregel: 5k bis 10k CPU-Zyklen/Switch (Bereich ms)

Kontextwechsel

Definition

Kontextwechsel nennt man den Vorgang, bei dem die Bearbeitung des aktuellen Threads unterbrochen und zu einem anderen Thread gewechselt wird.
Dabei wird der Kontext (im Wesentlichen die Prozessor-Register) des aktuellen Threads gesichert und der Kontext des neuen restauriert.

Kontextwechsel...

- ... sind nicht kostenfrei
- ... brauchen selbst CPU-Zeit
- ... und verbrauchen somit Ressourcen

Die tatsächlichen Kosten sind plattformabhängig.

Daumenregel: 5k bis 10k CPU-Zyklen/Switch (Bereich ms)

Kontextwechsel

Definition

Kontextwechsel nennt man den Vorgang, bei dem die Bearbeitung des aktuellen Threads unterbrochen und zu einem anderen Thread gewechselt wird.
Dabei wird der Kontext (im Wesentlichen die Prozessor-Register) des aktuellen Threads gesichert und der Kontext des neuen restauriert.

Kontextwechsel...

- ... sind nicht kostenfrei
- ... brauchen selbst CPU-Zeit
- ... und verbrauchen somit Ressourcen

Die tatsächlichen Kosten sind plattformabhängig.

Daumenregel: 5k bis 10k CPU-Zyklen/Switch (Bereich ms)

Kontextwechsel

Definition

Kontextwechsel nennt man den Vorgang, bei dem die Bearbeitung des aktuellen Threads unterbrochen und zu einem anderen Thread gewechselt wird.
Dabei wird der Kontext (im Wesentlichen die Prozessor-Register) des aktuellen Threads gesichert und der Kontext des neuen restauriert.

Kontextwechsel...

- ... sind nicht kostenfrei
- ... brauchen selbst CPU-Zeit
- ... und verbrauchen somit Ressourcen

Die tatsächlichen Kosten sind plattformabhängig.

Daumenregel: 5k bis 10k CPU-Zyklen/Switch (Bereich ms)

Speichersynchronisierung

- `synchronized` und `volatile` garantieren exakte Reihenfolge
- Intern werden dafür sog. Speicherbarrieren verwendet
- Speicherbarrieren verhindern Kompileroptimierung bzw. Re-ordering
- JVM kann „uncontended synchronization“ weiter optimieren...
- ... escape analysis: z.B. „lock elision“, „lock coarsening“

Speichersynchronisierung

- `synchronized` und `volatile` garantieren exakte Reihenfolge
- Intern werden dafür sog. Speicherbarrieren verwendet
- Speicherbarrieren verhindern Kompileroptimierung bzw. Re-ordering
- JVM kann „uncontended synchronization“ weiter optimieren...
- ... escape analysis: z.B. „lock elision“, „lock coarsening“

Speichersynchronisierung

- `synchronized` und `volatile` garantieren exakte Reihenfolge
- Intern werden dafür sog. Speicherbarrieren verwendet
- Speicherbarrieren verhindern Kompileroptimierung bzw. Re-ordering
- JVM kann „uncontended synchronization“ weiter optimieren...
- ... escape analysis: z.B. „lock elision“, „lock coarsening“

Speichersynchronisierung

- `synchronized` und `volatile` garantieren exakte Reihenfolge
- Intern werden dafür sog. Speicherbarrieren verwendet
- Speicherbarrieren verhindern Kompileroptimierung bzw. Re-ordering
- JVM kann „uncontended synchronization“ weiter optimieren...
- ... escape analysis: z.B. „lock elision“, „lock coarsening“

Speichersynchronisierung

- `synchronized` und `volatile` garantieren exakte Reihenfolge
- Intern werden dafür sog. Speicherbarrieren verwendet
- Speicherbarrieren verhindern Kompileroptimierung bzw. Re-ordering
- JVM kann „uncontended synchronization“ weiter optimieren...
- ... escape analysis: z.B. „lock elision“, „lock coarsening“

Speichersynchronisierung

- `synchronized` und `volatile` garantieren exakte Reihenfolge
- Intern werden dafür sog. Speicherbarrieren verwendet
- Speicherbarrieren verhindern Kompileroptimierung bzw. Re-ordering
- JVM kann „uncontended synchronization“ weiter optimieren...
- ... escape analysis: z.B. „lock elision“, „lock coarsening“

Blocking

- Wenn ein Lock umstritten ist, muss ein Thread blocken
- JVM kennt spin-waiting und suspending
- Effizienz hängt von Kontextwechsel-Overhead und Wartezeit ab

spin-waiting

effizient für kurze Wartezeit

suspending

effizient für lange Wartezeit

Blocking

- Wenn ein Lock umstritten ist, muss ein Thread blocken
- JVM kennt spin-waiting und suspending
- Effizienz hängt von Kontextwechsel-Overhead und Wartezeit ab

spin-waiting

effizient für kurze Wartezeit

suspending

effizient für lange Wartezeit

Blocking

- Wenn ein Lock umstritten ist, muss ein Thread blocken
- JVM kennt spin-waiting und suspending
- Effizienz hängt von Kontextwechsel-Overhead und Wartezeit ab

spin-waiting

effizient für kurze Wartezeit

suspending

effizient für lange Wartezeit

Reduzierung der „lock contention“

Sequentialisierung verschlechtert Skalierbarkeit,

Kontextwechsel verschlechtern Performanz

- Contended locking verursacht beides (Sequentialisierung und Kontextwechsel)
- Contended locking kann vermieden werden durch
  - Reduzierung der Haltezeit (eines Locks)
  - Reduzierung der Lockfrequenz
  - Ersetzen von exclusiven locks (intrinsic $\rightarrow$ explicit)

Reduzierung der „lock contention“

Sequentialisierung verschlechtert Skalierbarkeit,

Kontextwechsel verschlechtern Performanz

- Contended locking verursacht beides (Sequentialisierung und Kontextwechsel)
- Contended locking kann vermieden werden durch
  - Reduzierung der Haltezeit (eines Locks)
  - Reduzierung der Lockfrequenz
  - Ersetzen von exclusiven locks (intrinsic $\leftrightarrow$ explicit)

Reduzierung der „lock contention“

Sequentialisierung verschlechtert Skalierbarkeit,

Kontextwechsel verschlechtern Performanz

- Contended locking verursacht beides (Sequentialisierung und Kontextwechsel)
- Contended locking kann vermieden werden durch
  - Reduzierung der Haltezeit (eines Locks)
  - Reduzierung der Lockfrequenz
  - Ersetzen von exclusiven locks (intrinsic $\leftrightarrow$ explicit)

Reduzierung der „lock contention“

Sequentialisierung verschlechtert Skalierbarkeit,

Kontextwechsel verschlechtern Performanz

- Contended locking verursacht beides (Sequentialisierung und Kontextwechsel)
- Contended locking kann vermieden werden durch
  - Reduzierung der Haltezeit (eines Locks)
  - Reduzierung der Lockfrequenz
  - Ersetzen von exclusiven locks (intrinsic ↔ explicit)

Reduzierung der „lock contention“

Sequentialisierung verschlechtert Skalierbarkeit,

Kontextwechsel verschlechtern Performanz

- Contended locking verursacht beides (Sequentialisierung und Kontextwechsel)
- Contended locking kann vermieden werden durch
  - Reduzierung der Haltezeit (eines Locks)
  - Reduzierung der Lockfrequenz
  - Ersetzen von exclusiven locks (intrinsic $\leftrightarrow$ explicit)

Reduzierung der „lock contention“

Sequentialisierung verschlechtert Skalierbarkeit,

Kontextwechsel verschlechtern Performanz

- Contended locking verursacht beides (Sequentialisierung und Kontextwechsel)
- Contended locking kann vermieden werden durch
  - Reduzierung der Haltezeit (eines Locks)
  - Reduzierung der Lockfrequenz
  - Ersetzen von exclusiven locks (intrinsic ↔ explicit)

Reduzierung der „lock contention“

- Locks so kurz wie möglich halten: „Get in, get out“
- Lock Granularität verringern: „Lock splitting“
- Set von Objekten anlegen und (noch) feiner aufteilen: „lock striping“
- Alternativen zu exclusiven Locks verwenden: z.B. ReadWriteLock (multiple-reader, single-writer)
- Kein pooling von Objekten: Java Garbage Collector

Reduzierung der „lock contention“

- Locks so kurz wie möglich halten: „Get in, get out“
- Lock Granularität verringern: „Lock splitting“
- Set von Objekten anlegen und (noch) feiner aufteilen: „lock striping“
- Alternativen zu exclusiven Locks verwenden: z.B. ReadWriteLock (multiple-reader, single-writer)
- Kein pooling von Objekten: Java Garbage Collector

Reduzierung der „lock contention“

- Locks so kurz wie möglich halten: „Get in, get out“
- Lock Granularität verringern: „Lock splitting“
- Set von Objekten anlegen und (noch) feiner aufteilen: „lock striping“
- Alternativen zu exclusiven Locks verwenden: z.B. ReadWriteLock (multiple-reader, single-writer)
- Kein pooling von Objekten: Java Garbage Collector

Reduzierung der „lock contention“

- Locks so kurz wie möglich halten: „Get in, get out“
- Lock Granularität verringern: „Lock splitting“
- Set von Objekten anlegen und (noch) feiner aufteilen: „lock striping“
- Alternativen zu exclusiven Locks verwenden: z.B. ReadWriteLock (multiple-reader, single-writer)
- Kein pooling von Objekten: Java Garbage Collector

Reduzierung der „lock contention“

- Locks so kurz wie möglich halten: „Get in, get out“
- Lock Granularität verringern: „Lock splitting“
- Set von Objekten anlegen und (noch) feiner aufteilen: „lock striping“
- Alternativen zu exclusiven Locks verwenden: z.B. ReadWriteLock (multiple-reader, single-writer)
- Kein pooling von Objekten: Java Garbage Collector

Was nehmen wir mit nach Hause?

- Performanz und Skalierbarkeit sind (meist) gegenläufig
- Sequentiellen Codeanteil vermeiden
- First make it run, then make it fast...
- Measure, don't guess...

Was nehmen wir mit nach Hause?

- Performanz und Skalierbarkeit sind (meist) gegenläufig
- Sequentiellen Codeanteil vermeiden
- First make it run, then make it fast...
- Measure, don't guess...

Was nehmen wir mit nach Hause?

- Performanz und Skalierbarkeit sind (meist) gegenläufig
- Sequentiellen Codeanteil vermeiden
- First make it run, then make it fast...
- Measure, don't guess...

Was nehmen wir mit nach Hause?

- Performanz und Skalierbarkeit sind (meist) gegenläufig
- Sequentiellen Codeanteil vermeiden
- First make it run, then make it fast...
- Measure, don't guess...

Frage 1

Frage:

Warum ist die synchronizedList nochmal explizit mit einem Lock geschützt?

Antwort:

„In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation.“[a]

[a] http://bmraoblog.blogspot.com/2006/11/using-collectionssynchronizedlist.html

Frage 1

Frage:

Warum ist die synchronizedList nochmal explizit mit einem Lock geschützt?

Antwort:

„In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation.“[a]

[a] http://bmraoblog.blogspot.com/2006/11/using-collectionssynchronizedlist.html