Sven Osterwald

12.05.2010

Concurrent Objects

Proseminar

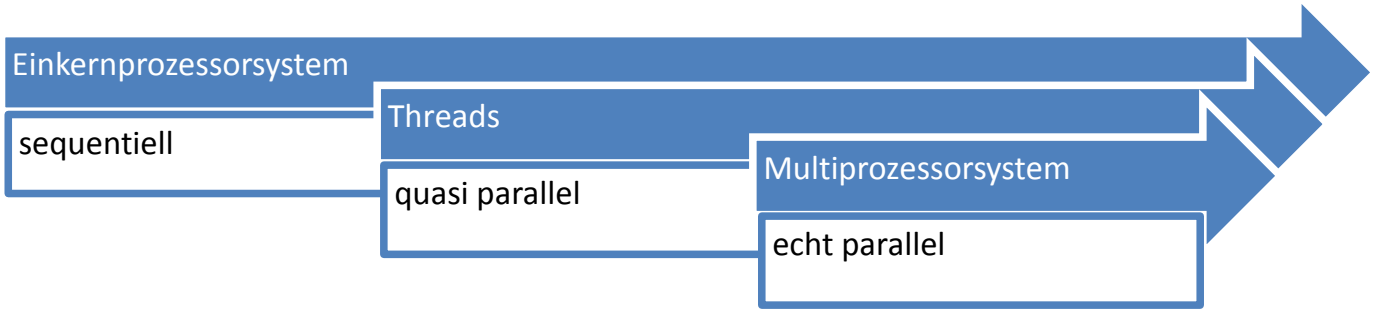
Parallele Programmierung in Java

Überblick

1. Einführung
2. Beispiel FIFO-Queue mit und ohne Lock
3. Korrektheit bei sequentiellen Objekten
4. Korrektheit bei konkurrierenden Objekten
   - Quiescent Consistency
   - Sequential Consistency
   - Linearizability
5. Nebenläufigkeit in Java
6. Zusammenfassung

1. Einführung

- Historie



- Nebenläufige Datenstrukturen sind Objekte, die Daten in einer Multiprozessorumgebung speichern und organisieren
- Es müssen Sicherheitseigenschaften (safety properties) und Lebensdauereigenschaften (liveness properties) erfüllt sein.
- Will man nebenläufige Strukturen untersuchen, versucht man die Ausführung sequentiell abzubilden.

Bsp. FIFO-Queue mit Lock

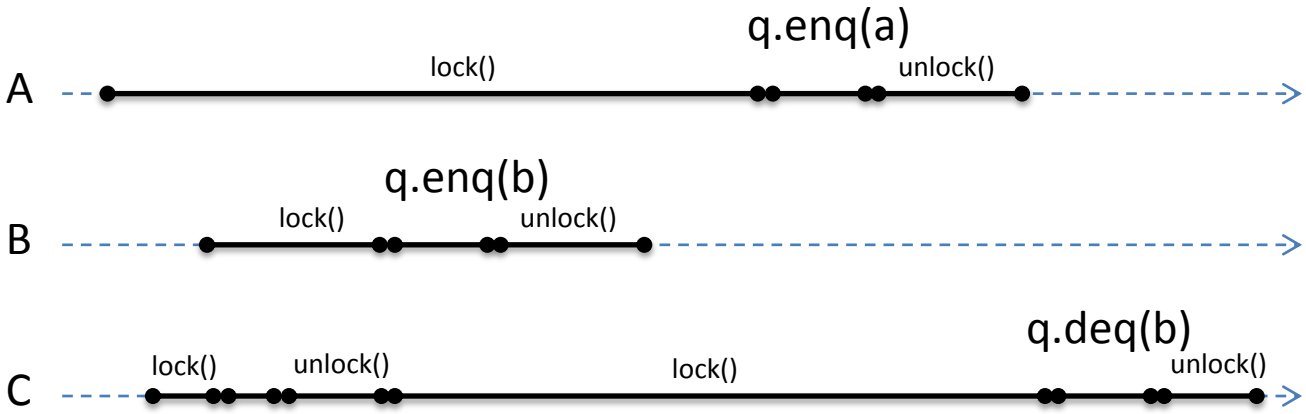
```
class LockBasedQueue<T> {
   int head, tail;
   T[] items;
   Lock lock;

   public LockBasedQueue(int capacity) {
      head = 0; tail = 0;
      lock = new Lock();
      items = (T[])new Object[capacity];
   }

   public void enq(T x) throws FullException {
      lock.lock();
      try {
         if (tail - head == items.length)
            throw new FullException();
         items[tail % items.length] = x;
         tail++;
      } finally {
         lock.unlock();
      }
   }
```

- Methoden enq() und deq()
- Lock

```
   public T deq() throws EmptyException {
      lock.lock();
      try {
         if (tail == head)
            throw new EmptyException();
         T x = items[head % items.length];
         head++;
         return x;
      } finally {
         lock.unlock();
      }
   }
}
```

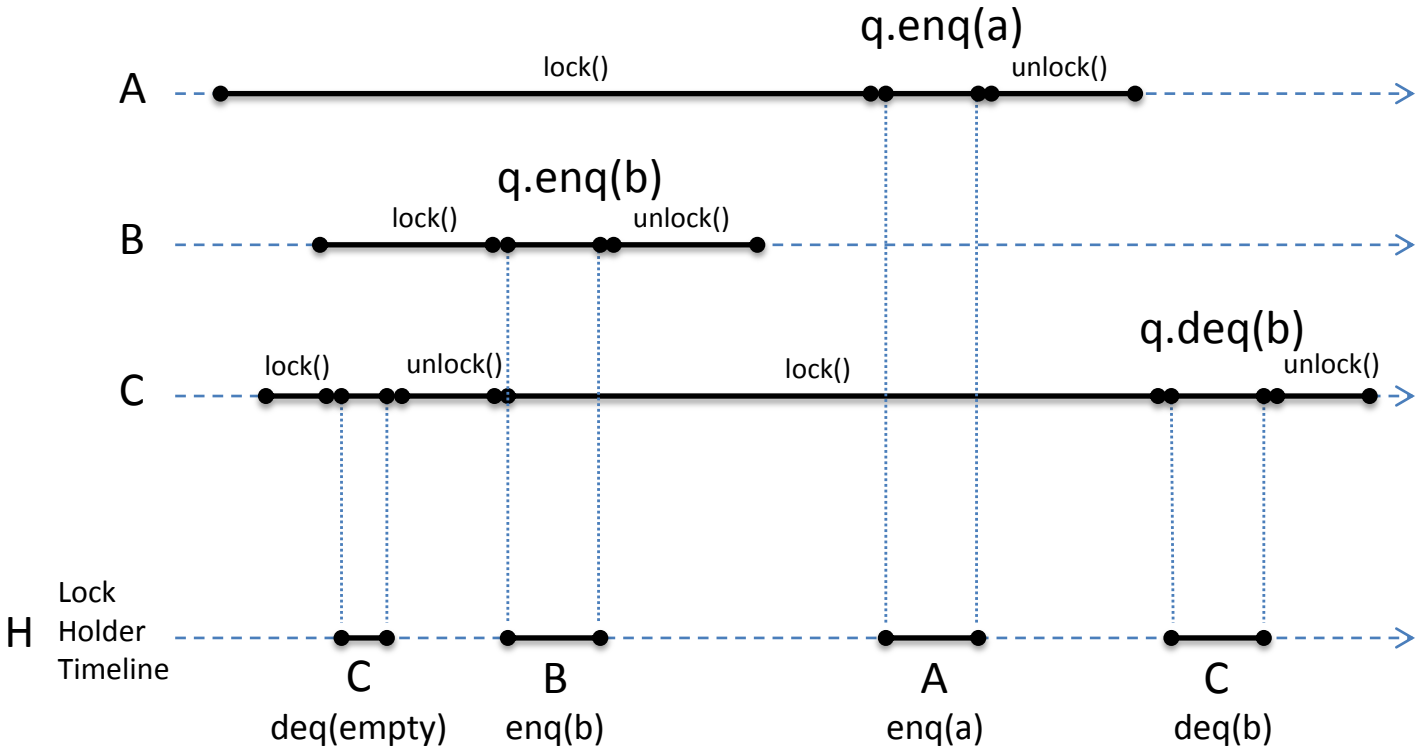
Bsp. FIFO-Queue mit Lock

Beispielablauf:



Bsp. FIFO-Queue mit Lock

Beispielablauf:



Korrektheit bei sequentiellen Objekten

- Jedes Objekt befindet sich in einem Status, der nur über seine Methoden verändert werden kann

- Bsp.: q.enq(z) ergibt q*z (Konkatenation)
  bei a*q ergibt q.deq() den Status q

- sequential specification: es gibt immer einen Zustand zwischen zwei Aufrufen

Bsp. FIFO-Queue ohne Lock

```
class WaitFreeQueue<T> {
   volatile int head = 0, tail = 0;
   T[] items;

   public WaitFreeQueue(int capacity) {
      items = (T[])new Object[capacity];
      head = 0; tail = 0;
   }

   public void enq(T x) throws FullException {
      if (tail - head == items.length)
         throw new FullException();
      items[tail % items.length] = x;
      tail++;
   }

   public T deq() throws EmptyException {
      if (tail - head == 0)
         throw new EmptyException();
      T x = items[head % items.length];
      head++;
      return x;
   }
}
```

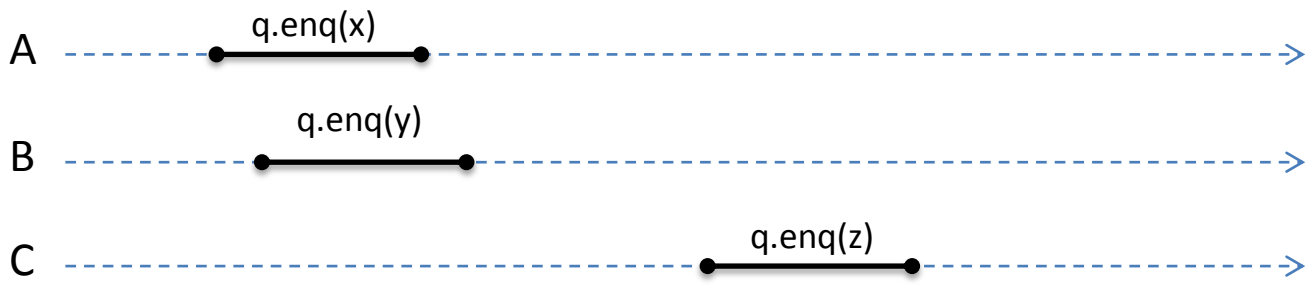
- single enqueuer / single dequeuer
- identisch mit LockBasedQueue
- ohne lock

Korrektheit bei nebenläufigen Objekten

- keine feste Reihenfolge der Abarbeitung, Unterbrechungen jederzeit möglich
- verwirrender Zustand: überlappende Zugriffe
- Korrektheitseigenschaften für konkurrierende Datenobjekte zum Nachweis des korrekten Verhaltens, z.B.
  - Quiescent Consistency
  - Sequential Consistency
  - Linearizability

Quiescent Consistency

- **Beispiel**:



- Ein Zugriff besteht aus einem Aufruf (invocation) und
  einer Antwort (response)

- **Definition** pending (unvollständig):

Ist in einer Ablauffolge das Antwortereignis eines Zugriffs noch nicht passiert, ist dieser Zugriff unvollständig.

- **Definition** quiescent (still):

Ein Objekt ist still, wenn es nicht mehr unvollständig ist.

Quiescent Consistency

- **Prinzip 1**:

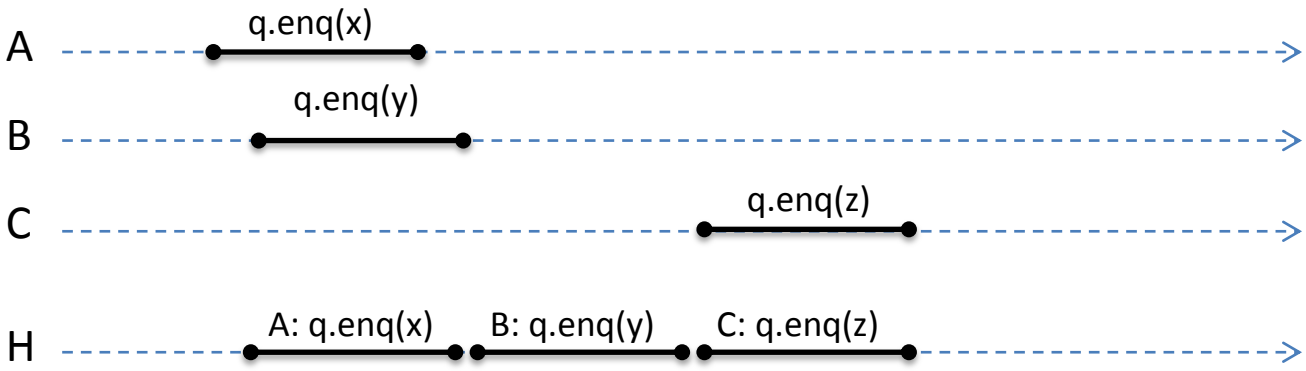
Zugriffe sollten
(1) in einer sequentiellen Reihenfolge
(2) und immer isoliert geschehen (nur ein Zugriff auf einmal)

- **Prinzip 2**:

Zugriffe, die durch eine Periode der Stille getrennt werden, sollten Auswirkungen auf die wirkliche Reihenfolge haben.

Quiescent Consistency

- **Beispiel**: Reihenfolge von x und y unbekannt, aber z zuletzt



- **Definition** non blocking (nicht blockierend)

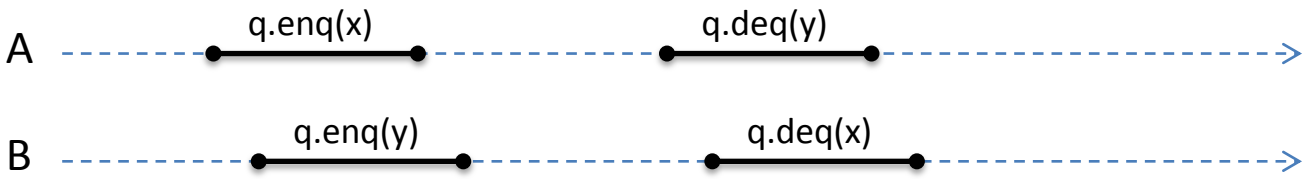
Es muss kein Zugriff blockiert werden, um auf einen anderen zu warten bis dieser fertig ist.

- **Definition** compositional (zusammensetzbar)

Aus Eigenschaft P gilt für alle Einzelteile folgt P gilt für das gesamte System.

Sequential Consistency

- Beispiel:

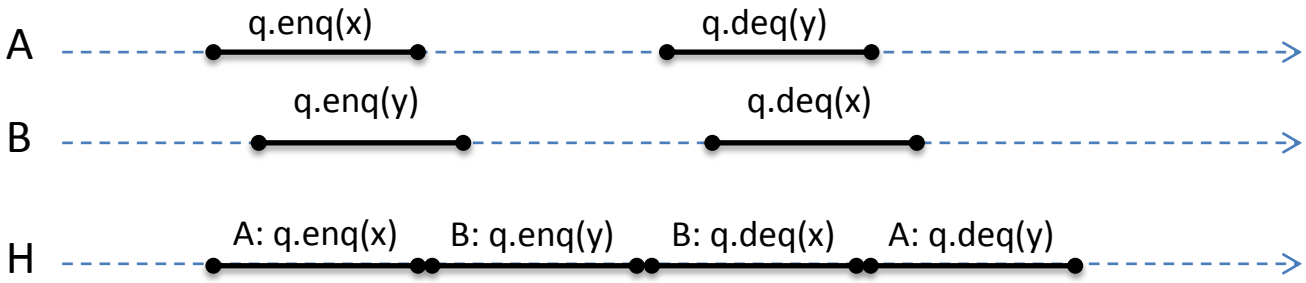


- **Prinzip**:

Eine Ablauffolge ist sequentiell konsistent, wenn sie zu einer sequentiellen Ablauffolge äquivalent ist. Die zeitliche Ordnung von nicht nebenläufigen Zugriffen muss dabei nicht erhalten bleiben.

Sequential Consistency

- Beispiel: sequentiell konsistente Ausführung

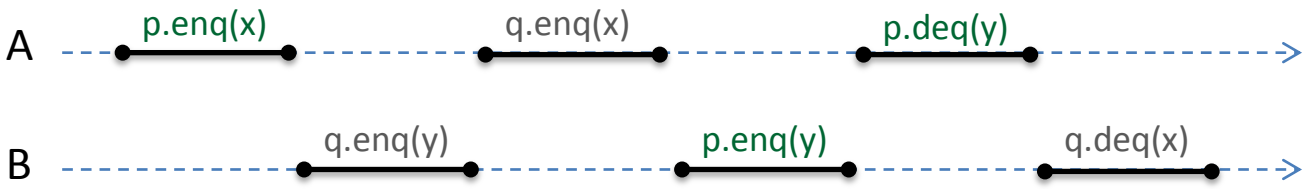


- Zum Nachweis ordnen wir die Zugriffe, sodass sie
  - mit einer sequentiellen Reihenfolge übereinstimmen
  - die sequentielle Spezifikation des Objekts treffen
  - die Programmordnung eines Threads aufrecht erhalten

- Es existieren zwei mögliche sequentielle Ordnungen:
  - A fügt x ein, B fügt y ein, B entfernt x, A entfernt y
  - B fügt y ein, A fügt x ein, A entfernt y, B entfernt x

Sequential Consistency

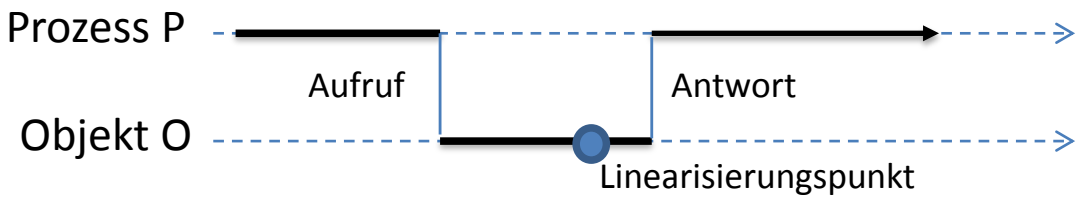
- non blocking und non compositional
- Beispiel: zusammengesetzte sequentielle Inkonsistenz



  - zwei Queue-Objekte p und q
  - beide sequentiell konsistent
  - zusammen nicht sequentiell konsistent (non compositional)

Linearizability

- Linearisierungspunkt eines Zugriffs

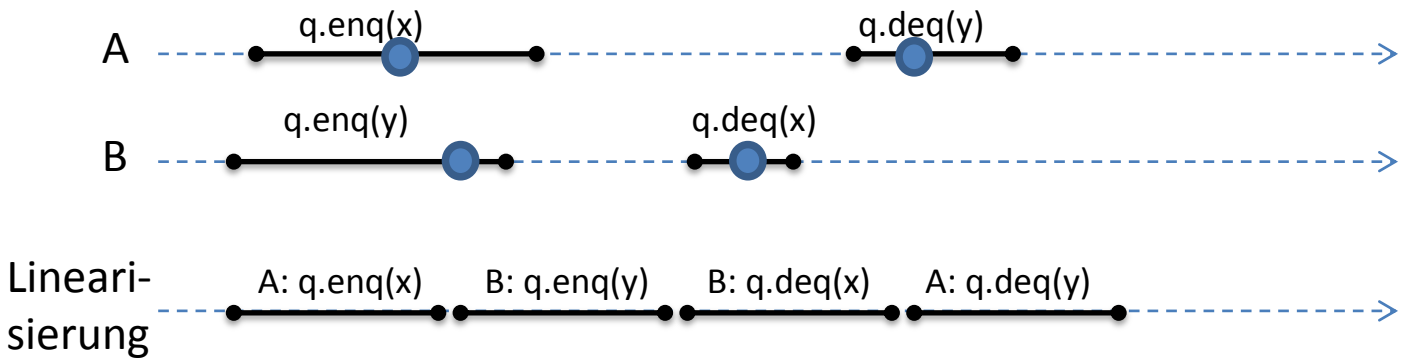


- **Prinzip**:

Eine Ablauffolge ist linearisierbar, wenn zu jedem Zugriff ein Zeitpunkt während dieses Zugriffes, der Linearisierungspunkt, existiert, so dass der Zugriff zu exakt diesem Zeitpunkt eine atomare Zustandstransition bewirkt.

Linearizability

- Beispiel: linearisierbare Ablauffolge und Linearisierung



- schließt sequentielle Konsistenz ein
- Linearisierungspunkt bei q.deq(): head++ bzw. EmptyException
- non blocking und compositional

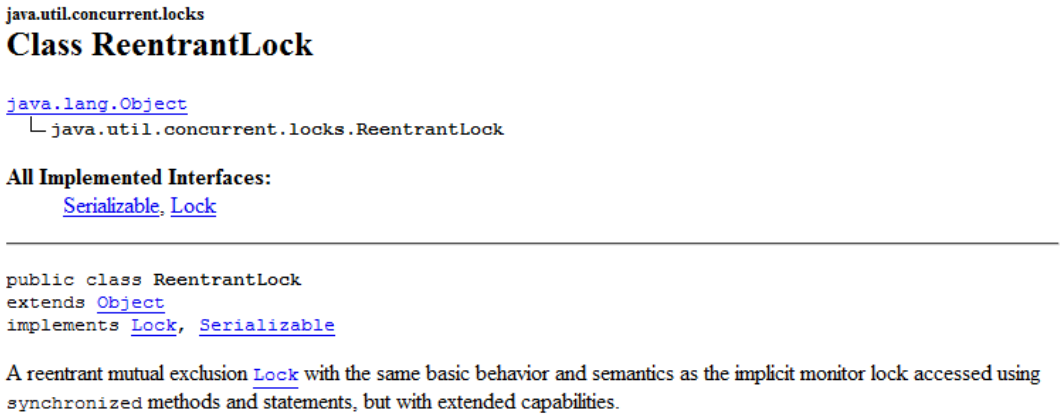
Nebenläufigkeit in Java

- Java garantiert keine Linearisierbarkeit oder sequentielle Konsistenz (wegen Compileroptimierungen)

- Es gibt dennoch Möglichkeiten sequentielle Konsistenz zu erreichen
  - Locks and synchronized-Blocks
  - volatile Fields
  - final Fields

Locks and synchronized-Blocks

- Ein Thread erreicht mutual exclusion, wenn er einen `synchronized`-Block oder -Methode betritt.
- Gleiche Wirkung mit ReentrantLock aus dem java.util.concurrent.locks package



java.util.concurrent.locks

Class ReentrantLock

java.lang.Object
  └ java.util.concurrent.locks.ReentrantLock

All Implemented Interfaces:
Serializable, Lock

```
public class ReentrantLock
extends Object
implements Lock, Serializable
```

A reentrant mutual exclusion `Lock` with the same basic behavior and semantics as the implicit monitor lock accessed using `synchronized` methods and statements, but with extended capabilities.

- Es wird Linearisierbarkeit garantiert, wenn ein Objekt mit diesen Mitteln geschützt wird

volatile Fields

- Schlüsselwort

  `volatile`

  zeigt an, dass Variablen auch außerhalb des aktuellen Threads verändert werden können

- Der Wert der Variablen wird jedes Mal aus dem Speicher genommen und in ihn geschrieben

- linearisierbar

final Fields

- Variablen mit dem Schlüsselwort

  `final`

  dürfen nach ihrer Initialisierung nicht mehr verändert werden

- Der korrekte Wert der Variablen ist für alle anderen Threads ohne zusätzliche Synchronisation sichtbar

Zusammenfassung

- Zentrales Prinzip der Korrektheitseigenschaften:
  Sie bringen nebenläufige Zugriffe in eine sequentielle Reihenfolge

Korrektheitseigenschaft	non blocking	compositional
quiescent consistency	+	+
sequential consistency	+	-
linearizability	+	+

- In Java lässt sich Linearisierbarkeit mit verschiedenen Methoden erreichen (`synchronized`, `volatile`, `final`)

Quellen

[1] Herlihy, M. & Shavit, N. (2008). The Art Of Multiprocessor Programming. Morgan Kaufmann Publishers.

[2] Krainz, J. (2009). Linearisierbarkeit als Korrektheitseigenschaft für nicht-blockierende Datenstrukturen. Department Informatik FAU Erlangen-Nürnberg.

[3] Krüger, G. & Stark, T. (2009). Handbuch der Java Programmierung. Addison Wesley.

Vielen Dank für Eure Aufmerksamkeit!