# Nebenläufige und verteilte Programme CS2301


Th. Letschert

TH Mittelhessen Gießen

*University of Applied Sciences*


## Monitorprogramme: Passive Monitore und aktive Threads

- Monitor-Anwendungen: Spezifikation und Implementierung
- Leser-Schreiber Synchronisation
- Problematik der Monitorprogramme: Deadlocks
- Lock-Bereiche

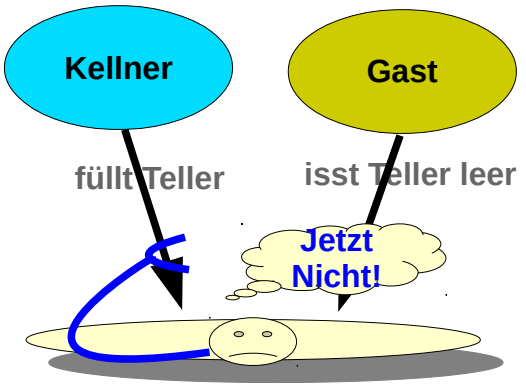
## Monitor

*Konzept von Tony Hoare zur Strukturierung von Anwendungen mit Synchronisationsbedarf.*

*Basis / Inspiration der Mechanismen der Synchronisation der JVM*

### Prinzipen

- Die Resource (!) sorgt für ihre korrekte Benutzung – nicht die Threads welche die Ressource nutzen
- Die eingesetzten Mittel sind: Pro Ressource
  - ein Mutex, sowie
  - Bedingungsvariablen nach Bedarf




### Umsetzung

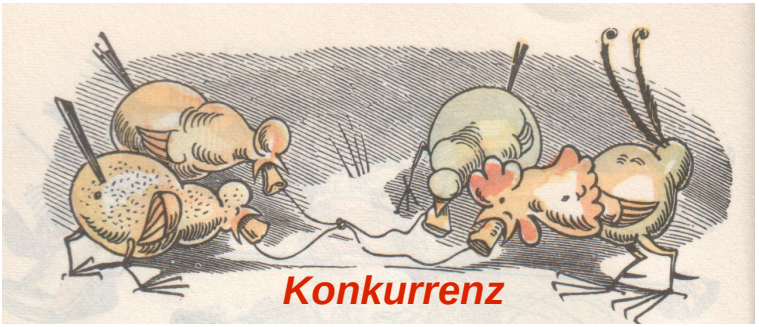
- Als Muster / Programm-Idiom
- Als Sprachfeature: Klasse plus Synchronisation.
  Java war die erste gängige Programmiersprache
  mit einer Unterstützung der nebenläufigen Programmierung
  durch Threads und ein integriertes Monitorkonzept.
- Monitore in Scala unterscheiden sich nicht wesentlich von ihrer Java-Variante

# Monitore

***Klassisches Konzept zur Strukturierung von Anwendungen mit Synchronisationsbedarf***

## Anwendungsfeld

**(Systemnahe) Anwendungen mit (wenigen) Threads / Prozessen. Diese greifen auf gemeinsame Ressourcen zu.**




## Nebenläufigkeit

**dient in erster Linie nicht der Beschleunigung sondern der Strukturierung der Anwendung**

## Synchronisation in Monitoren

- **Behandle Konkurrenzsituationen:**
  **Gegenseitiger Ausschluss**
- **Ermögliche Kooperation:**
  **Bedingungssynchronisaton**

# Monitor

## Struktur einer Monitor-Anwendung

- **Aktive Nutzer (Threads / Prozesse) nutzen**
- **Monitore: Passive Ressourcen (kein Thread / kein Prozess), die selbst für ihre korrekte Nutzung sorgen**


*Paradigmatische Monitor-Anwendung: Produzent-Konsument mit Puffer*

***Handlungsfäden begegnen sich nur in (Methoden von) Monitoren. Nur hier gibt es einen Bedarf an Synchronisation. Nur hier werden Probleme der Synchronisation gelöst***

# Monitor-Anwendungen

## Spezifikation

*Keine Implementierung ohne Spezifikation*

**Spezifikation von Monitor-Klassen:**

**Klasseninvarinate ~definiert~> Implementierung des gegenseitigen Ausschluss'**

**Die Datenstrukturen sind vor und nach dem Aufruf einer öffentlichen Methode in korrektem / konsistentem Zustand: welche Zustände sind korrekt / konsistent?**

**=> Notwendigen gegenseitigen Ausschluss realisieren: synchronized**

**Vorbedingung ~definiert~> Implementierung der Bedinungssynchronisation**

**Voraussetzung der Benutzung einer Methode.**

**=> Notwendige Bedingungssynchronisation realisieren: wait / notify**

## Spezifikation

### *Beispiel Puffer: Spezifikation*

```
class Puffer {

  private var inhalt: Double = 0.0
  private var voll: Boolean = false;

  // die Variablen müssen konsistente Werte haben
  // hier:
  // voll = true gdw. der Wert von inhalt nach der
  //        letzten Zuweisung noch nicht gelesen wurde
  // voll = false gdw. der Wert von inhalt nach der
  //        letzten Zuweisung gelesen wurde

  // Aufruf nur wenn voll == false
  def fuelle(x: Double): Unit = {
    inhalt = x;
    voll = true;
  }

  // Aufruf nur wenn voll == true
  def leere(): Double = {
    voll = false;
    inhalt;
  }

}
```

***Kein inkonsistenter Zustand:***

***Eine halb ausgeführte Zuweisung an die double-Variable (nicht atomar!) muss ausgeschlossen werden!***
***Ein „falscher Wert“ der Variablen voll durch eine halb ausgeführte Methode muss ausgeschlossen werden.***

***Beachte Vorbedingung:***

***Jeder geschriebene Wert muss gelesen werden. Jeder gelesene Wert muss geschrieben worden sein.***

## Implementierung

***Beispiel Puffer: Implementierung als Monitor***

```
class Puffer {
  private var inhalt: Double = 0.0
  private var voll: Boolean = false;

  def fuelle(x: Double): Unit = synchronized {
    while (voll) wait();
    inhalt = x;
    voll = true;
    notifyAll();
  }

  def leere(): Double = synchronized {
    while (!voll) wait();
    voll = false;
    notifyAll();
    inhalt;
  }
}
```

**An allen Stellen, an der eine Bedingung erwartet wird:**

```
while(!Bedingung) wait();
```

**An allen Stellen an denen eine Bedingung positiv beeinflusst wird:**

```
notify() / notifyAll()
```

# Monitor-Anwendungen

## notify / notifyAll

Die Bedingungsvariable, auf die sich notify / notifyAll beziehen, hat eine Warteschlange

In der Warteschlange warten eventuell gleichzeitig mehrere Threads auf unterschiedliche Bedingungen

Ein notify weckt dann eventuell den Falschen: das notify geht verloren

In dem Fall sollte notifyAll eingesetzt werden: Alle werden geweckt, einer kommt dran

```
class Puffer {
  private var inhalt: Double = 0.0
  private var voll: Boolean = false;

  def fuelle(x: Double): Unit = synchronized {
    while (voll) wait();
    inhalt = x;
    voll = true;
    notify();
  }

  def leere(): Double = synchronized {
    while (!voll) wait();
    voll = false;
    notify();
    inhalt;
  }
}
```

Inter welchen Umständen kann statt notifyAll das (effizientere) notify eingesetzt werden?

# Monitor-Anwendungen

## notify vs. notifyAll

**notify weckt eventuell den Falschen .. und der Richtige schläft weiter: Deadlock !**

**notify reaktiviert (irgendeinen) der wartenden Threads**

- **OK falls gilt:**
  **wenn immer nur einer wartet, oder**
  **wenn alle auf die gleiche Bedingung warten**
- **Nicht OK falls gilt:**
  **wenn mehrere Threads auf *unterschiedliche Bedingungen* warten**
  **Es *kann* passieren, dass der geweckte (wait()-Ausführende) Thread nicht auf die Bedingung wartet, die der Weckende (notify-Aufrufer) erfüllt hat**

```
class Puffer {
  private var inhalt: Double = 0.0
  private var voll: Boolean = false;

  def fuelle(x: Double): Unit = synchronized {
    while (voll) wait();
    inhalt = x;
    voll = true;
    notify();
  }

  def leere(): Double = synchronized {
    while (!voll) wait();
    voll = false;
    notify();
    inhalt;
  }
}
```

**Ein notifyAll kann durch das (effizientere) notify ersetzt werden, wenn alle Threads in der Warteschlange auf die gleiche Bedingung warten. Der erste in der Schlange darum immer zu Recht geweckt wurde.**

**Hier beispielsweise bei abwechselnder Nutzung, also wenn es genau einen Füller und genau einen Leerer gibt.**

# Leser-Schreiber-Synchronisation

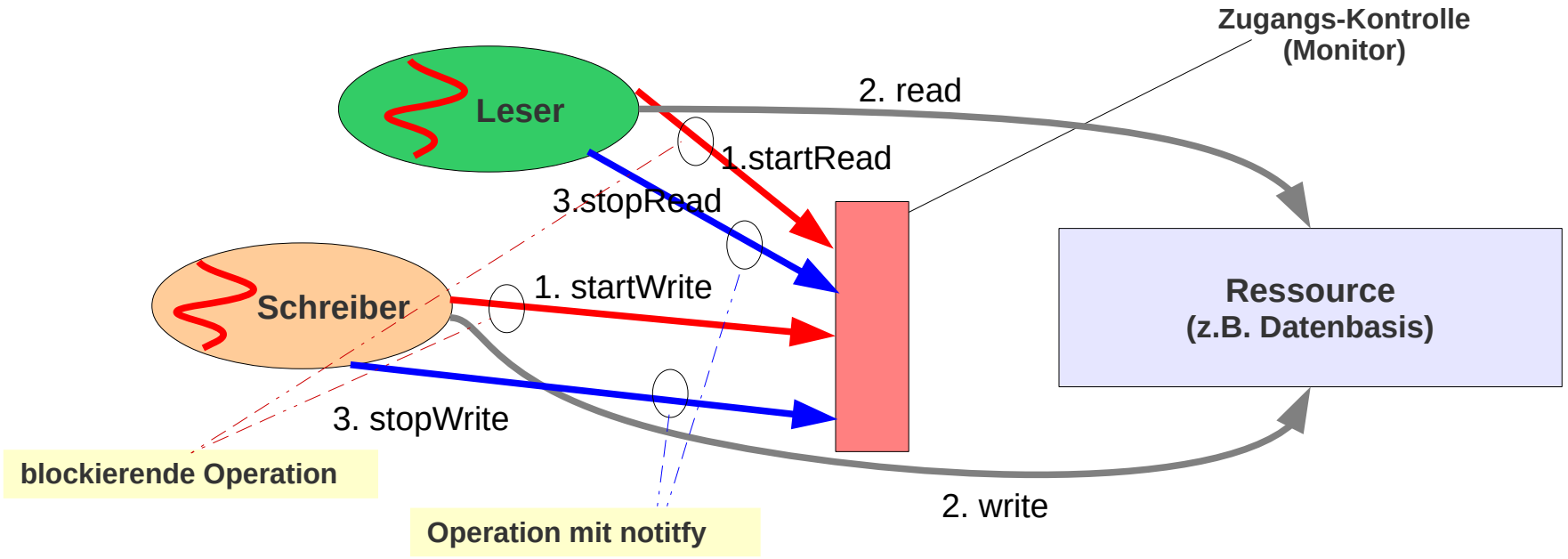
## Aufgabenstellung

**Eine Ressource wird auf zwei Arten benutzt**

- **Leser dürfen in unbeschränkter Zahl gleichzeitig zugreifen**
- **Schreiber benötigen exklusiven Zugriff:**
  **kein anderer Schreiber oder Leser ist gleichzeitig zugelassen**

## Lösung

**Der Zugang zur Ressource wird durch einen Monitor überwacht, die Ressource selbst ist unsynchronisiert**



## Leser-Schreiber-Synchronisation

### Protokoll der Nutzer

- Lesen: startRead read stopRead
- Schreiben: startWrite write stopWrite

### Bedingungssynchronisation im Monitor

zähle die aktiven Schreiber und Leser

### Zutrittsbedingung:

- Leser: warte wenn es Schreiber gibt
- Schreiber: warte wenn es Schreiber oder Leser gibt

## Leser-Schreiber-Synchronisation

```
trait Ressource[T] {
  def read(): T
  def write(x: T): Unit
}
```

```
class ReadWriteMonitor[T](r: Ressource[T]) {

  private var count_readers = 0;
  private var count_writers = 0;


  def write(x: T): Unit = {
    startWrite
    r.write(x);
    stopWrite
  }

  def read(): T = {
    startRead
    val v = r.read();
    stopRead
    v
  }


}
```

```
private def startWrite: Unit = synchronized {
    while ((count_readers > 0) || (count_writers > 0)) wait();
    count_writers = count_writers + 1;
  }

  private def stopWrite: Unit = synchronized {
    count_writers = count_writers -1
    if ( count_writers == 0) notifyAll();
  }

  private def startRead: Unit = synchronized {
    while (count_writers > 0) wait ();
    count_readers = count_readers+1;
  }

  private def stopRead: Unit = synchronized {
    count_readers = count_readers-1
    if (count_readers == 0)
    notifyAll();
  }
```

## Leser-Schreiber-Synchronisation

### Ressource ist eine Abbildung Schlüssel → Wert

### SW-Struktur

- Kontroll-Code: abstrakte Basisklasse
- Ressourcen-Code: Implementierung abstrakter Methoden

```
abstract class AbstractSynchronizedMap[K, V] {
  private var count_readers = 0;
  private var count_writers = 0;

  def write(key: K, value: V): Unit = {
    startWrite
    writeImpl(key, value)
    stopWrite
  }

  def read(key: K): V = {
    startRead
    val v = readImpl(key);
    stopRead
    v
  }

  protected def writeImpl(key: K, value: V): Unit
  protected def readImpl(key: K): V

  private def startWrite: Unit = synchronized { … }
  private def stopWrite: Unit = synchronized { … }
  private def startRead: Unit = synchronized { … }
  private def stopRead: Unit = synchronized { … }
}
```

oder als Trait

```
trait AbstractSynchronizedMap[K, V]
```

falls in Mehrfachvererbung eingesetzt

## Beispiel: Cache mit Leser/Schreiber-Synchronisation – 1

```
class Cache[K, V](val size: Int)(implicit val mk: Manifest[K], implicit val mv: Manifest[V]) {

  private val keys = new Array[K](size)      // Cached keys
  private val values = new Array[V](size);   // Cached values
  private val hits  = new Array[Int](size)   // Cache-Hits for this entry

  def get(k: K): Option[V] = {
    for(i <- 0 until size) {
      if ( (keys(i) != null) && keys(i).equals(k) ){
        hits(i) = hits(i)+1
        return Some(values(i))
      }
    }
    None
  }

  def put(k: K, v: V): Unit = {
    val victim: Int = hits.zipWithIndex.min._2
    keys(victim) = k
    values(victim) = v
    hits(victim) = 1
   }
}
```

***Mit den Manifest-Parametern wird es möglich Arrays mit generischen Elementtyp anzulegen.***

## Beispiel: Cache mit Leser/Schreiber-Synchronisation – 2

```
class SynchronizedCache[K, V](val size:Int)(
      implicit val mk: Manifest[K],
      implicit val mv: Manifest[V]) extends AbstractSynchronizedMap[K, V] {

  private val cache = new Cache[K,V](size);

  override protected def readImpl(k: K): V = cache.get(k).getOrElse(
        throw new Exception
      )

  override protected def writeImpl(k: K, v: V): Unit = cache.put(k, v);

}
```

*Ein Cache mit Leser-Schreiber-Synchronisation*

```
class SynchronizedCache[K, V](override val size: Int)(
      override implicit val mk: Manifest[K],
      override implicit val mv: Manifest[V]
      ) extends Cache[K, V](size)(mk, mv)
        with AbstractSynchronizedMap[K, V]  {

      override def readImpl(k: K): V = get(k).getOrElse(
        throw new Exception
       )

      override def writeImpl(k: K, v: V): Unit = put(k, v)
 }
```

*… oder so mit Mehrfachvererbung – wer's mag*

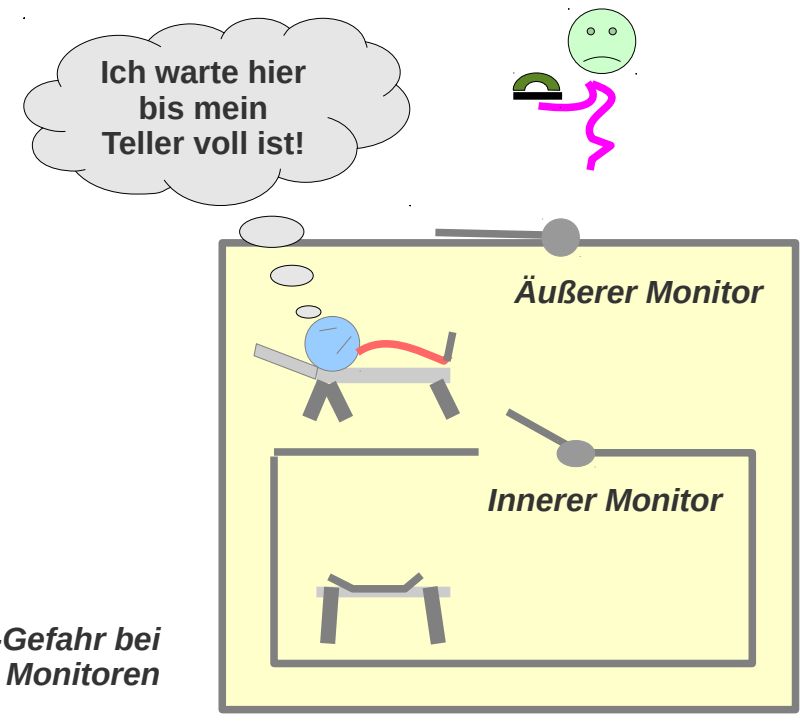
# Geschachtelte und wieder-betretende Monitore

## Wiederbetretender Monitor

Ein Thread kann sich mehrfach in dem gleichen Monitor befinden:
Aufruf einer Monitor-Methode aus einer Monitor-Methode des gleichen Monitors
Java erlaubt das Wieder-betreten (Lock) eines schon betretenen Monitors

## Geschachtelter Monitor

Ein Thread kann sich gleichzeitig in mehreren Monitoren befinden:
Aufruf einer Monitor-Methode eines anderen Monitors aus einer Monitor-Methode
Gefahr von Deadlocks




*Deadlock-Gefahr bei geschachtleten Monitoren*

## *Monitorprogramme*: komplex und fehleranfällig

### Deadlocks/ Verklemmungen

Monitorprogramme enthalten blockierende Aktionen und Synchronisationsanweisungen

Deren Zusammenspiel ist oft nicht leicht zu überschauen

Probleme wie *Deadlocks* können in vielfälliger und oft unvorhergesehener Art auftreten

### Lazy Val, object und Synchronisationsprobleme

**lazy val**: ein Wert der beim ersten Zugriff und maximal einmal berechnet wird.

**object**: eine Singleton-Klasse

Beide werden maximal einmal initialisiert.

Um Wettbewerbssituationen zu vermeinen, wird dabei ***double checked locking Muster**** eingesetzt

Bei gegenseitigen Abhängigkeiten kann es dabei zu *Deadlocks* kommen

* Zu diesem Muster siehe beispielsweise:
Angelika Langer *Java Memory Model, volatile und das Double-Check-Idiom*
http://www.angelikalanger.com/Articles/EffectiveJava/41.JMM-DoubleCheck/41.JMM-DoubleCheck.html

## Beispiel: Deadlock bei object / lazy-val Initialisierung

```
object LazyVal_Main extends App {

  lazy val x: Int = {
    val t : Thread = new Thread(
           () => println(s"initializing $x")
    })

    t start()
    t join()

    42
  }

  println(x)
}
```

***Deadlock:***

***Main-Thread greift auf x zu***
***x  wird initialisiert, dabei wird ein lock auf x gesetzt.***
***Dann wird t gestartet, t will x lesen, da x gelockt ist, muss t auf die Freigabe des Locks warten.***

***Deadlock: Main-Thread wartet auf t, t wartet auf Main-Thread***

***In Shared-Memory-Programmen lauern Deadlocks und andere Probleme an den unvorhergesehensten Ecken ...***

## Sicherheit und Lebendigkeit (Safety, Liveness)

Thema für einen Masterkurs

**Synchronisation hat zwei Ziele:**

- **Böses verhindern**
- **Gutes ermöglichen**

**Allgemeiner spricht man bei einer Anwendung von Sicherheit und Lebendigkeit**

- **Sicherheit**

  **Sicherheit ist die Abwesenheit von Bösem**

  **Bei nebenläufigen Anwendungen hat das Böse im Wesentlichen zwei Gesichter:**

  - **Interferenz der Threads: Threads die sich in die Quere kommen, indem ein kritischer Abschnitt von mehr als einem Thread ausgeführt wird.**
  - **Deadlocks: gegenseitiges endloses Warten**

- **Lebendigkeit**

  **Mit Lebendigkeit ist gemeint, dass das Ziel der Anwendung irgendwie irgendwann erreicht wird:**

  - **Eine Anfrage wird tatsächlich irgendwann beantwortet,**
  - **eine gesendete Nachricht kommt irgendwann an,**
  - **ein Thread der einen kritischen Abschnitt betreten will, betritt ihn irgendwann**
  - **...**

- **Das Begriffspaar wurde von Leslie Lamport* eingeführt bei der Diskussion von Korrektheitsbeweisen von nebenläufigen Programmen.**

*Leslie Lamport: *Proving the correctness of multiprocessor programs.* IEEE Transactions on Software Engineering, 3(2) Seite 125-143, März 1977

# Monitor-Anwendungen

## Lock-Bereiche

**Setze Locks so dass der gelockte Bereich so kurz wie möglich und so weit wie nötig**

**Beispiel**

```
object CachingSqrt_OK {
  private var lastX: Double = -1.0;
  private var lastR: Double = -1.0;
  private var hit: Int = 0;

  def sqrt(x: Double): Double = synchronized {
    if ( x < 0 ) throw new ArithmeticException();
    if ( x == lastX ) {
      hit = hit + 1; lastR;
    } else {
      lastX = x; lastR = heron(x); lastR;
    }
  }

  private def heron(x: Double): Double = {
    var a = 1.0;
    for (i <- 0 to 1000000)
      a = (a + x/a)/2;
     a
  }
}
```

***Lang andauernde Operationen (hier heron) sollten nicht in einem gelockten Bereich ablaufen!***

```
object CachingSqrt_Good {
  private var lastX: Double = -1.0;
  private var lastR: Double = -1.0;
  private var hit: Int = 0;

  def sqrt(x: Double): Double = {
    if ( x < 0 ) throw new ArithmeticException();
    var r: Double = -1.0;
    synchronized {
      if ( x == lastX ) {
        hit = hit+1; r = lastR;
      }
    }
    if ( r == -1.0 ) {
      r = heron(x);
      synchronized {
        lastX = x; lastR = r;
      }
    }
    r
  }

  private def heron(x: Double): Double = {
    var a = 1.0;
    for (i <- 0 to 1000000)
      a = (a + x/a)/2;
     a
  }
}
```