

Algorithmen & Datenstrukturen

# Synchronisation

Vertiefung von O16\_IP\_Synchronisation

Roger Diehl



# Inhalt

- Elementare Synchronisationsmechanismen
- Synchronisation durch Monitor-Konzept
- Fallstricke
- Zusammenfassung

# Lernziele

- Sie kennen die Konzepte für den Zugriff auf gemeinsame Ressourcen und können diese anwenden.
- Sie kennen den erweiterten Thread Lebenszyklus und können bestimmen, wann ein Thread sich in welchem Zustand befindet.
- Sie kennen das Monitor Konzept, wissen was Reentrant Monitore sowie Nested Monitore sind und können diese vermeiden.
- Sie können das Monitor Konzept in Java anwenden.
- Sie kennen verschiedene Fallstricke bei der Verwendung von **synchronized** und können diese vermeiden.

# **Elementare Synchronisationsmechanismen**

# Zugriff auf gemeinsame Ressourcen

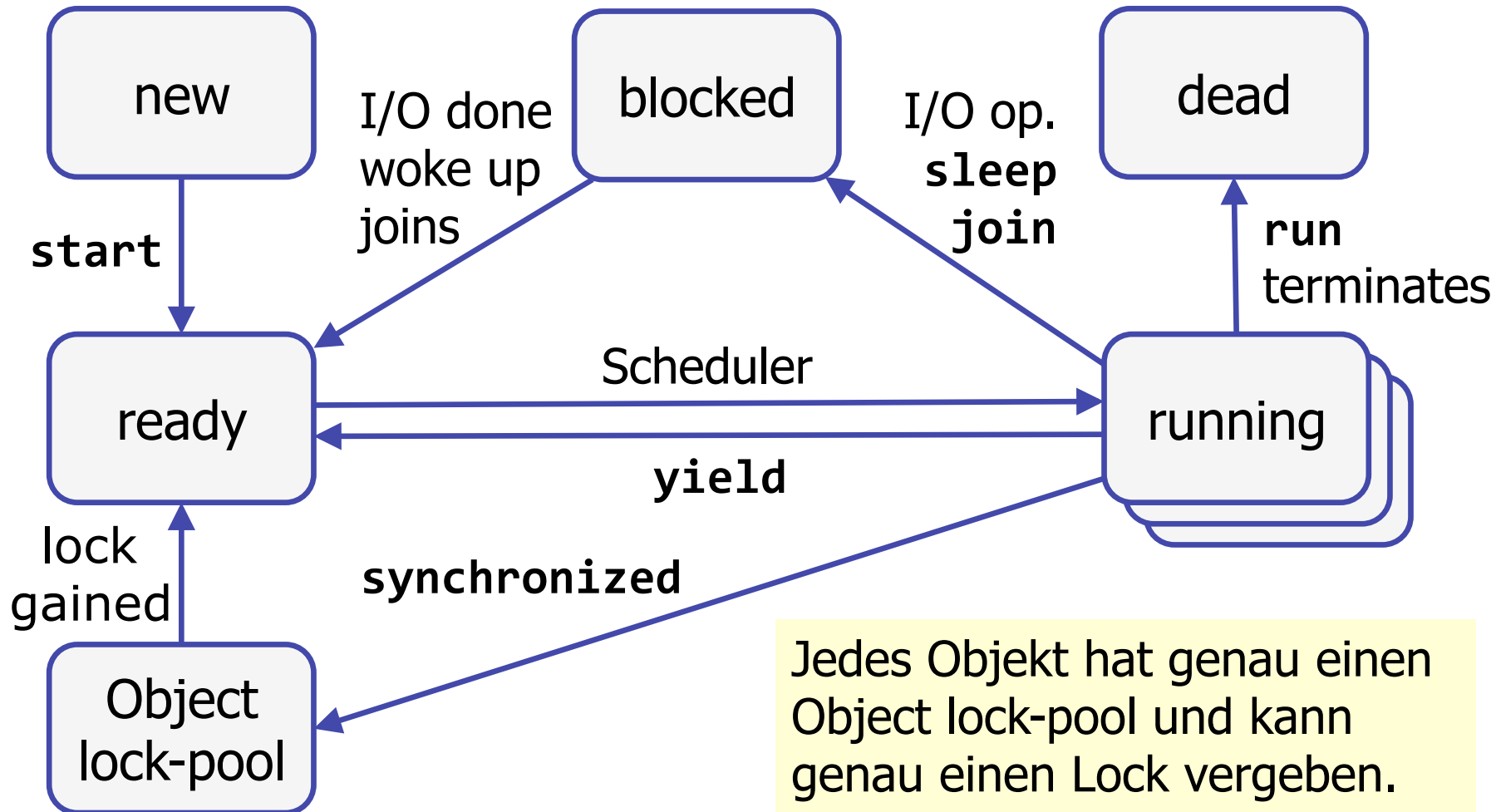
Eine gute Lösung für den gegenseitigen Ausschluss muss vier Bedingungen erfüllen:

1. In einem **kritischen Abschnitt** darf sich zu jedem Zeitpunkt höchstens immer **nur ein Thread** befinden.
2. Es dürfen **keine Annahmen** über die zugrunde liegende Hardware (Clock, CPU-Anzahl etc.) gemacht werden.
3. Ein Thread darf **andere Threads nicht blockieren**, ausser er ist in einem kritischen Bereich.
4. Es muss sichergestellt sein, dass ein Thread **nicht unendlich lange warten** muss, bis er in den kritischen Bereich eintreten kann.

[Andrew S. Tanenbaum 1994: Moderne Betriebssysteme]

# Thread Lebenszyklus

Ein Thread kann während seines Lebenszyklus die folgenden Zustände (vereinfacht) einnehmen:



# Schlüsselwort `synchronized`

- Alle Java-Objekte, sowohl herkömmliche Instanzen als auch Klassenobjekte, besitzen einen impliziten Lock.
- Den Zugriff auf den Lock erhält man durch das Schlüsselwort **`synchronized`**.
- Während des Wartens auf den Lock kann der Thread nicht (mit **`interrupt`**) unterbrochen werden.

```
class Test {
```

```
    int count;
```

```
    synchronized void bump() {  
        count++;
```

```
    }
```

```
    static int classCount;
```

```
    static synchronized void classBump() {  
        classCount++;
```

```
    }
```

```
}
```

Quelle: Java Language Specification 8.0

Lock für das Objekt **`this`**

Lock der Klasse **`Test`**

# Schlüsselwort `synchronized`

- Dieser Code ist äquivalent zur vorhergehenden Seite.

```
public class BumpTest {
```

Quelle: Java Language Specification 8.0

```
    int count;  
    void bump() {  
        synchronized (this) {  
            count++;  
        }  
    }
```

Lock für das Objekt `this`

```
    static int classCount;  
    static void classBump() {  
        try {  
            synchronized (Class.forName("BumpTest")) {  
                classCount++;  
            }  
        } catch (ClassNotFoundException e) {  
        }  
    }  
}
```

Lock für die Klasse `BumpTest`

die Klasse `BumpTest`  
stellt den Lock-Pool



# Beispiel: Thread-sicheres Singleton

- Das Beispiel zeigt eine Implementierung des Singleton-Patterns mit verzögerter Initialisierung (lazy instantiation).

```
public final class Singleton {  
    private static Singleton instance;  
    private Singleton() {  
        // komplizierte Initialisierung  
    }  
    public static synchronized Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
    // weitere Objektmethoden  
}
```

Codeskizze

Zugriff auf die Singleton-Instanz ist durch die Klasse **Singleton** geschützt.

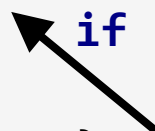
Der Zugriff auf die Singleton-Instanz ist hier **nur sequentiell** möglich!

## Beispiel: Thread-sicheres Singleton (optimiert)

- Eine Optimierung der Zugriffszeit ist das Double-Checked-Locking.
- Wird das Objekt angelegt, muss sichergestellt sein, dass nur ein Thread den Konstruktor aufruft.
- Der lesende Zugriff (**return** instance) auf die Singleton-Instanz muss nicht geschützt werden, sodass der Zugriff schnell ist.

```
private static volatile Singleton instance;
```

```
public static Singleton getInstance() {  
    if (instance == null) {  
        synchronized (Singleton.class) {  
            if (instance == null) {  
                instance = new Singleton();  
            }  
        }  
    }  
    return instance;  
}
```



Die Erzeugung der Singleton-Instanz ist geschützt.

Codeskizze

**volatile**  
garantiert, dass  
der Zustand  
immer aktuell ist,  
weil ein refresh  
auf dem Cache  
ausgeführt wird.

# Praxistipp

- Ein Lazy-Instantiation-Singleton gibt es auch ohne explizite Synchronisierung.

```
public final class SingletonNoSync {
```

Codeskizze

```
// Holder-Klasse für Singleton
```

```
private static class SingletonHolder {
```

```
    public static final SingletonNoSync instanceHolder =  
        new SingletonNoSync();
```

```
}
```

```
public static SingletonNoSync getInstance() {
```

```
    return SingletonHolder.instanceHolder;
```

```
}
```

```
private SingletonNoSync() {
```

```
    super();
```

```
    // komplizierte Initialisierung
```

```
}
```

```
}
```

Beim ersten Aufruf wird die Klasse **SingletonHolder** geladen.

# **Synchronisation durch Monitor-Konzept**

# Monitorkonzept in Java

- siehe OOP Input → **O16\_IP\_Synchronisation; Folien 14-18**
- Schon in den 70er-Jahren haben Hansen und Hoare das Monitorkonzept eingeführt, um Multithreading sicher zu machen.
- Ein Monitor wäre auf Java übertragen eine spezielle Klasse mit folgenden Eigenschaften:
  - Alle Daten der Klasse müssen `private` deklariert sein.
  - Nur ein Thread kann zu jedem Zeitpunkt in einem Monitor aktiv sein.
  - Die Sperre kann eine beliebigen Anzahl von Bedingungen besitzen.
  - Es ist die Aufgabe der VM, den wechselseitigen Ausschluss der Monitoreingänge zu garantieren.
- Dieses allgemeine Konzept wird in Java nicht gänzlich übernommen und führt zum unsicheren Umgang, denn:
  - Attribute einer Klasse müssen bei Java nicht `private` sein.
  - Nicht alle Methoden müssen als `synchronized` deklariert sein.

# Mehrere Monitore

- Das Betreten von kritischen Abschnitten kann mit
  - wieder betretener Monitor (Reentrant Monitor) oder
  - geschachtelter Monitor (Nested Monitor)passieren.
- **reentrant**: eine synchronisierte Methode ruft eine andere auf, wobei beide synchronisierten Code Abschnitte denselben Objekt lock-Pool verwenden
- **nested**: eine synchronisierte Methode ruft eine andere auf, bei der beide synchronisierte Code Abschnitte verschiedene Objekt lock-Pools verwenden

# Reentrant Monitor

- Betritt ein Thread eine synchronisierte Methode/Abschnitt, dessen Lock er schon besitzt, kann er sofort eintreten ohne den lock-pool zu durchlaufen.
- Ohne diese Möglichkeit würden Rekursion nicht funktionieren!
- Dies reduziert aber auch die Parallelität, weil kritische Abschnitte künstlich vergrößert werden!
- Ein Reentrant Monitor kann geschwindigkeitssteigernd sein, wenn viele synchronisierte Methoden hintereinander aufgerufen werden. Muss aber nicht (z.B. bei **StringBuffer**).

# Nested Monitore

- Ein Thread kann beliebig viele Locks ergreifen.
- Geschachtelte (nested) Monitore führen leicht zu Verklemmungen (Deadlocks).
- Um die Deadlock-Gefahr zu vermeiden, sollte man in einer synchronisierten Methode oder einem synchronisierten Block niemals die Steuerung an den Client übergeben.
  - Das heisst: In einem synchronisierten Abschnitt nie eine öffentliche oder geschützte Methode aufrufen, die dazu da ist, überschrieben zu werden.



# Fallstricke

# Non-volatile-Zugriff auf gemeinsam benutzte Daten

```
public final class ModuloCounter {
```

```
    private int count = 0;  
    private final int mod;
```

Ohne **synchronized** muss **count** mit **volatile** deklariert werden.

```
    public ModuloCounter(final int mod) {  
        this.mod = mod;  
    }
```

```
    public int getValue() {  
        return count;  
    }
```

**Achtung:** Beim Aufruf von **getValue** wird aber kein refresh des Caches des Aufrufers durchgeführt.

Atomarer, lesender Zugriff auf **count**

```
    public synchronized void increment() {  
        count = (count + 1) % mod;  
    }
```

```
    public synchronized void decrement() {  
        count = (count - 1 + mod) % mod;  
    }
```

```
}
```

Codeskizze

# Gemeinsam benutzte Daten sind nur partiell geschützt

- Ein anderer Thread kann durch den direkten Zugriff das Attribut **commonData** lesen bzw. ändern (mit nicht atomare Operation).
- Wenn nicht alle Methoden, die mit gemeinsamen Ressourcen arbeiten, **synchronized** sind, kann sehr leicht ein Fehler eintreten.

## Codeskizze

```
public final class Unsafe {
```

```
    int commonData;
```

Sichtbarkeit **default**, **public**, **protected** von Attributen – alles schlecht!

```
    public synchronized void inc() {  
        commonData++;  
    }
```

```
    public void doIt() {  
        // macht was mit commonData..  
    }
```

**public** Methode mit Zugriff auf **commonData** (lesend/schreibend) – sehr schlecht!

```
}
```

# Verklemmungen (Deadlock)

- Eine Verklemmung ist eine Situation, bei der Threads gegenseitig aufeinander warten und für immer blockiert bleiben.

```
public final class SimpleDeadlockDemo {  
  
    private final Object lock1 = new Object();  
    private final Object lock2 = new Object();  
    public void doSomething() {  
        synchronized (lock1) {  
            synchronized (lock2) {  
                // macht was...  
            }  
        }  
    }  
  
    public void calcResult() {  
        synchronized (lock2) {  
            synchronized (lock1) {  
                // macht auch was...  
            }  
        }  
    }  
}
```

Codeskizze

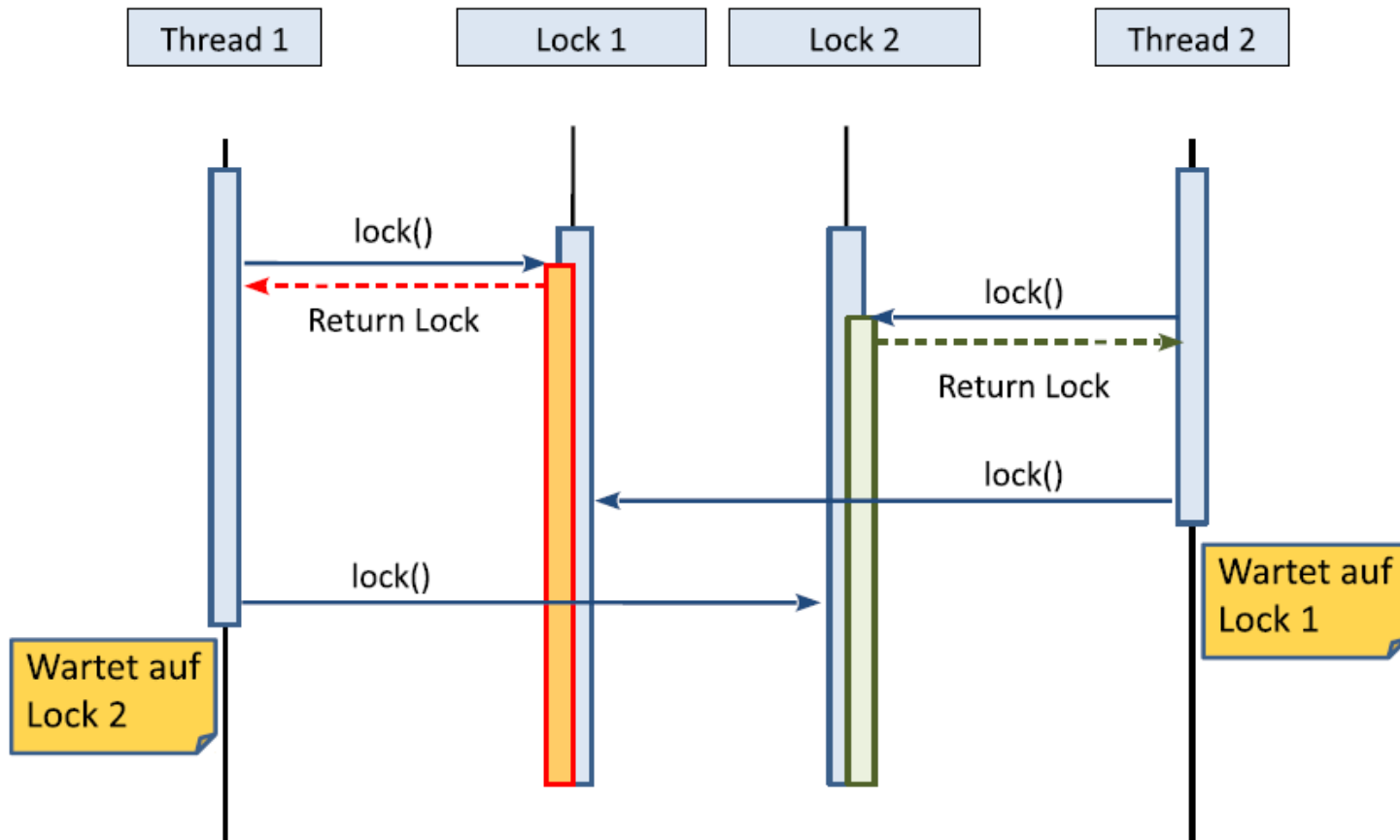
Ein Thread ruft  
nebenläufig  
**doSomething** auf.

Ein anderer Thread  
ruft nebenläufig  
**calcResult** auf.

**Achtung:** Nested Monitore haben  
immer Potential für Deadlocks.

# Deadlock durch gegenseitiges Warten

- Thread 1 ruft eine Methode mit dem Lock 1 auf, während Thread 2 nebenläufig eine Methode mit dem Lock 2 ausführt.
  - Dabei kann es zu einer Verklemmung kommen.



# Beispiel schwarzes Loch

- Die Klasse **BlackHole** stellt zwei Methoden **put** und **get** zur Verfügung und **queue** ist die gemeinsame Ressource.

```
public final class BlackHole {
```

```
    private final BlockingQueue<String> queue;
```

```
    public BlackHole() {  
        queue = new LinkedBlockingDeque<>();  
    }
```

```
    public synchronized void put(final String thing) {  
        queue.add(thing);  
    }
```

```
    public synchronized String get() throws InterruptedException {  
        return queue.take();  
    }  
}
```

Codeskizze

In einem schwarzen Loch kann etwas verschwinden oder etwas herauskommen...


# Beispiel schwarzes Loch – Deadlock

- Zwei Threads werden erzeugt und gestartet, der eine holt etwas aus dem schwarzen Loch, der andere wirft etwas hinein...


```
public final class DemoBlackhole {  
  
    public static void main(final String[] args) {  
        final BlackHole blackhole = new BlackHole();  
        System.out.println("Thread starts...");  
        new Thread(() -> {  
            try {  
                System.out.println(blackhole.get());  
            } catch (InterruptedException ex) {  
                System.err.println(ex.getMessage());  
            }  
        }, "Blackhole 'getter' thread").start();  
        new Thread(() -> {  
            blackhole.put("Sonne, Licht, irgendetwas...");  
        }, "Blackhole 'putter' thread").start();  
    }  
}
```

Codeskizze

Thread holt  
etwas heraus...



Thread gibt  
etwas hinein...



**Deadlock**

# Nested Monitore vermeiden!

- Das Beispiel **BlackHole** lässt sich einfach reparieren:
  - Die Methoden **put** und **get** nicht synchronisieren.
  - Die Klasse **BlockingQueue** ist thread-safe.
  - Falls eine Klasse thread-safe ist, muss dies dokumentiert sein.
- Eine fremde Methode, die ausserhalb eines synchronisierten Bereichs aufgerufen wird, bezeichnet man als offenen Aufruf.
  - [Lea 1999: Concurrent Programming in Java, 2.4.1.3]
- Offene Aufrufe verhindern nicht nur Deadlocks, sondern können auch die Nebenläufigkeit stark verbessern.
- **Regel:** Tun Sie möglichst wenig Arbeit in synchronisierten Bereichen. Vermeiden Sie übermässige Synchronisierung.
  - [Bloch 2002: Effektiv Java programmieren, Thema 49]



# Zusammenfassung

- Java realisiert mit dem Schlüsselwort **synchronized** ein Lock-Konzept (Ausschlussprinzip), mit dem Datenänderungen quasi atomar ausgeführt werden können.
- Jedes Objekt (einer Klasse) und jede Klasse besitzt hierzu einen impliziten Lock (mit lock-pool).
- Ein Thread darf eine **synchronized**-Methode bzw. einen -Block nur dann betreten, wenn er den zugeordneten Lock erhalten hat.
- Während des Wartens auf den Lock kann der Thread nicht (mit **interrupt**) unterbrochen werden.
- Durch den Einsatz von verschachtelten **synchronized**-Blöcke können Deadlocks entstehen.

**Fragen?**