

Synchronisation: Verteilter Zugriff auf gemeinsame Ressourcen

Carsten Gips (FH Bielefeld)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

Motivation: Verteilter Zugriff auf gemeinsame Ressourcen

```
public class Teaser implements Runnable {  
    private int val = 0;  
  
    public static void main(String... args) {  
        Teaser x = new Teaser();  
        new Thread(x).start();  
        new Thread(x).start();  
    }  
  
    private void incrVal() {  
        ++val;  
        System.out.println(Thread.currentThread().getId() + ": " + val);  
    }  
  
    public void run() {  
        IntStream.range(0, 5).forEach(i -> incrVal());  
    }  
}
```

Zugriff auf gemeinsame Ressourcen: Mehrseitige Synchronisierung

```
synchronized (<Object reference>) {  
    <statements (synchronized)>  
}
```

=> **“Mehrseitige Synchronisierung”**

Zugriff auf gemeinsame Ressourcen: Mehrseitige Synchronisierung

```
synchronized (<Object reference>) {  
    <statements (synchronized)>  
}
```

=> “Mehrseitige Synchronisierung”

```
private void incrVal() {  
    synchronized (this) { ++val; }  
}
```

Synchronisierte Methoden

```
void f() {  
    synchronized (this) {  
        ...  
    }  
}
```

=>

```
synchronized void f() {  
    ...  
}
```

Synchronisierte Methoden

```
void f() {  
    synchronized (this) {  
        ...  
    }  
}
```

=>

```
synchronized void f() {  
    ...  
}
```

```
private synchronized void incrVal() {  
    ++val;  
}
```

Probleme bei der (mehrseitigen) Synchronisierung: Deadlocks

```
public class Deadlock {  
    private final String name;  
  
    public synchronized String getName() { return name; }  
    public synchronized void foo(Deadlock other) {  
        System.out.format("%s: %s.foo() \n", Thread.currentThread().getName(), name);  
        System.out.format("%s: %s.name()\n", Thread.currentThread().getName(), other.getName());  
    }  
  
    public static void main(String... args) {  
        final Deadlock a = new Deadlock("a");  
        final Deadlock b = new Deadlock("b");  
  
        new Thread(() -> a.foo(b)).start();  
        new Thread(() -> b.foo(a)).start();  
    }  
}
```

Warten auf andere Threads: Einseitige Synchronisierung

Problem

- Thread T1 wartet auf Arbeitsergebnis von T2
- T2 ist noch nicht fertig

Mögliche Lösungen

1. Aktives Warten (Polling): Permanente Abfrage
 - Kostet unnötig Rechenzeit
2. Schlafen mit `Thread.sleep()`
 - Etwas besser; aber wie lange soll man idealerweise schlafen?
3. Warten mit `T2.join()`
 - Macht nur Sinn, wenn T1 auf das *Ende* von T2 wartet
4. **Einseitige Synchronisierung** mit `wait()` und `notify()`
 - Das ist DIE Lösung für das Problem :)

Einseitige Synchronisierung mit *wait* und *notify*

- **wait**: Warten auf Erfüllung einer Bedingung (Thread blockiert):

```
synchronized (obj) {    // Geschützten Bereich betreten
    while (!condition) {
        try {
            obj.wait(); // Thread wird blockiert
        } catch (InterruptedException e) {}
    }
    ...    // Condition erfüllt: Tue Deine Arbeit
}
```

=> Bedingung nach Rückkehr von `wait` erneut prüfen!

Einseitige Synchronisierung mit *wait* und *notify* (cnt.)

- **notify**: Aufwecken von wartenden (blockierten) Threads:

```
synchronized (obj) {  
    obj.notify();           // einen Thread "in" obj aufwecken  
    obj.notifyAll();        // alle Threads "in" obj wecken  
}
```

Synchronisierungsbedarf bei verteiltem Zugriff auf gemeinsame Ressourcen:

- Vorsicht mit konkurrierendem Ressourcenzugriff:
Synchronisieren mit `synchronized` => **Mehrseitige Synchronisierung**
- Warten auf Ereignisse mit `wait` und `notify/notifyAll` => **Einseitige Synchronisierung**

LICENSE



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.