

# Entwurf und Implementierung paralleler Programme

---

Prof. Dr. Rolf Hennicker

27.11.2006

# **Kapitel 4**

## **Monitore und wechselseitiger Ausschluss**

## 4.1 Interferenzen

Parallel ablaufende Prozesse können sich gegenseitig (störend) beeinflussen.

### Beispiel (Client/Server):

```
||TWOCLIENTS_SERVER = (a:CLIENT || b:CLIENT || a:SERVER || b:SERVER)/  
{a.call/a.request,b.call/b.request,a.reply/a.wait,b.reply/b.wait}.
```

Möglicher Ablauf:

```
a.call  
a.service  
b.call  
b.service  
a.reply  
b.reply
```

Interferenz zwischen a.service und a.reply

Die Interferenz kann zu Problemen führen, wenn die Operation “service“ einen globalen Datenbestand (z.B. globale Variable) des Servers benützt.

**Beispiel:**

“service“ erhöht den Wert einer globalen Server-Variablen X um 1.

“reply“ gibt den neuen Wert von X zurück.

Erhöhen um 1 wird durchgeführt unter Verwendung des Akkumulators ACC im Rechnerkern.

## Trace mit Interferenz:

Sei zunächst  $X=7$

a.call

a.(ACC=X) / ACC=7

a.(ACC=ACC+1) / ACC=8

b.call

b.(ACC=X) / ACC=7

b.(ACC=ACC+1) / ACC=8

a.(X=ACC) / X=8

b.(X=ACC) / X=8

a.reply X / Rückgabe 8

b.reply X / Rückgabe 8

Richtig wäre aber die Rückgabe 9 bei b.reply!

## Beachte:

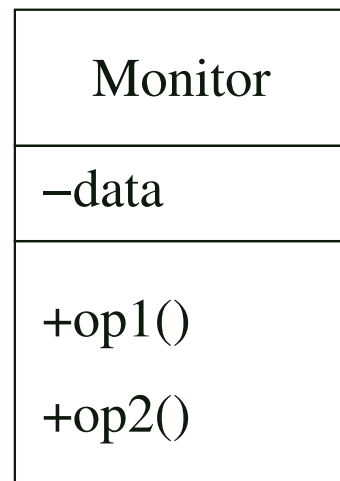
Operationen wie  $i++$  sind während der Ausführung eines Programmes *nicht* atomar und können zu Interferenzen führen!

## 4.2 Monitore

- Ein *Monitor* ist ein Objekt, das Daten verkapselt.
- Auf die Daten kann *nur* durch Operationen des Monitors zugegriffen werden.
- Zu jedem Zeitpunkt kann *nur eine* Ausführung einer Monitor-Operation aktiv sein.

### Konsequenz:

Die Manipulation der gekapselten Daten kann nur in *wechselseitigem Ausschluss* ("mutual exclusion") erfolgen. Es sind also keine Interferenzen möglich, die die Daten betreffen.



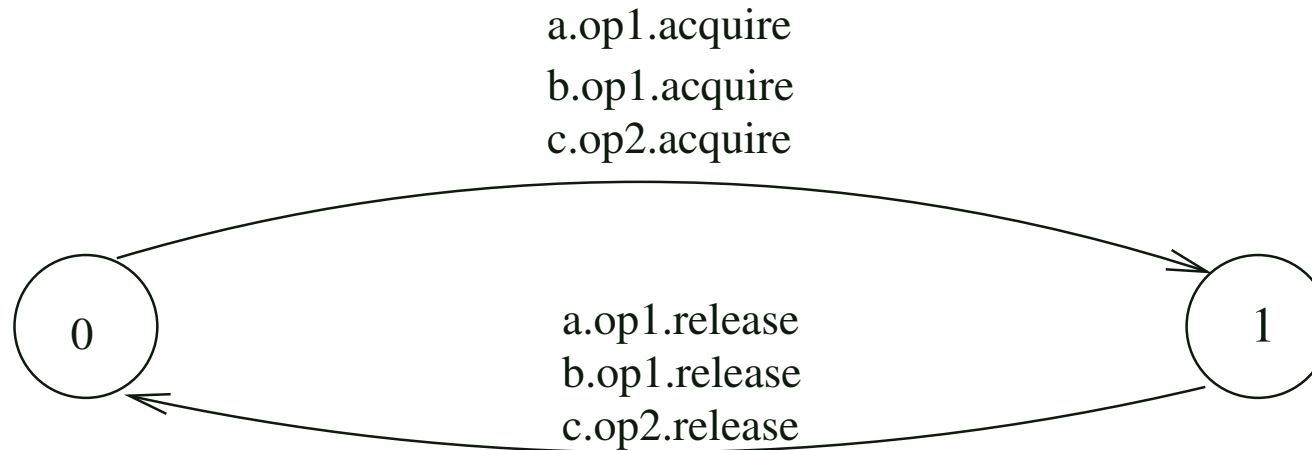
**Beispiel:**

Modellierung eines Monitors mit 2 Benutzern von op1 und einem Benutzer von op2.

$A = (a.do \rightarrow a.op1.call \rightarrow a.op1.returns \rightarrow A).$

$B = (b.do \rightarrow b.op1.call \rightarrow b.op1.returns \rightarrow B).$

$C = (c.do \rightarrow c.op2.call \rightarrow c.op2.returns \rightarrow C).$

**Beispieltrace:**LTS von  $\{a.op1, b.op1, c.op2\}::LOCK$ 

```

a.do
b.do
a.op1.call
b.op1.call
b.op1.acquire
c.do
c.op2.call
b.op1.body
b.op1.release
c.op2.acquire
b.op1.returns
  
```

## 4.3 Wechselseitiger Ausschluss in Java

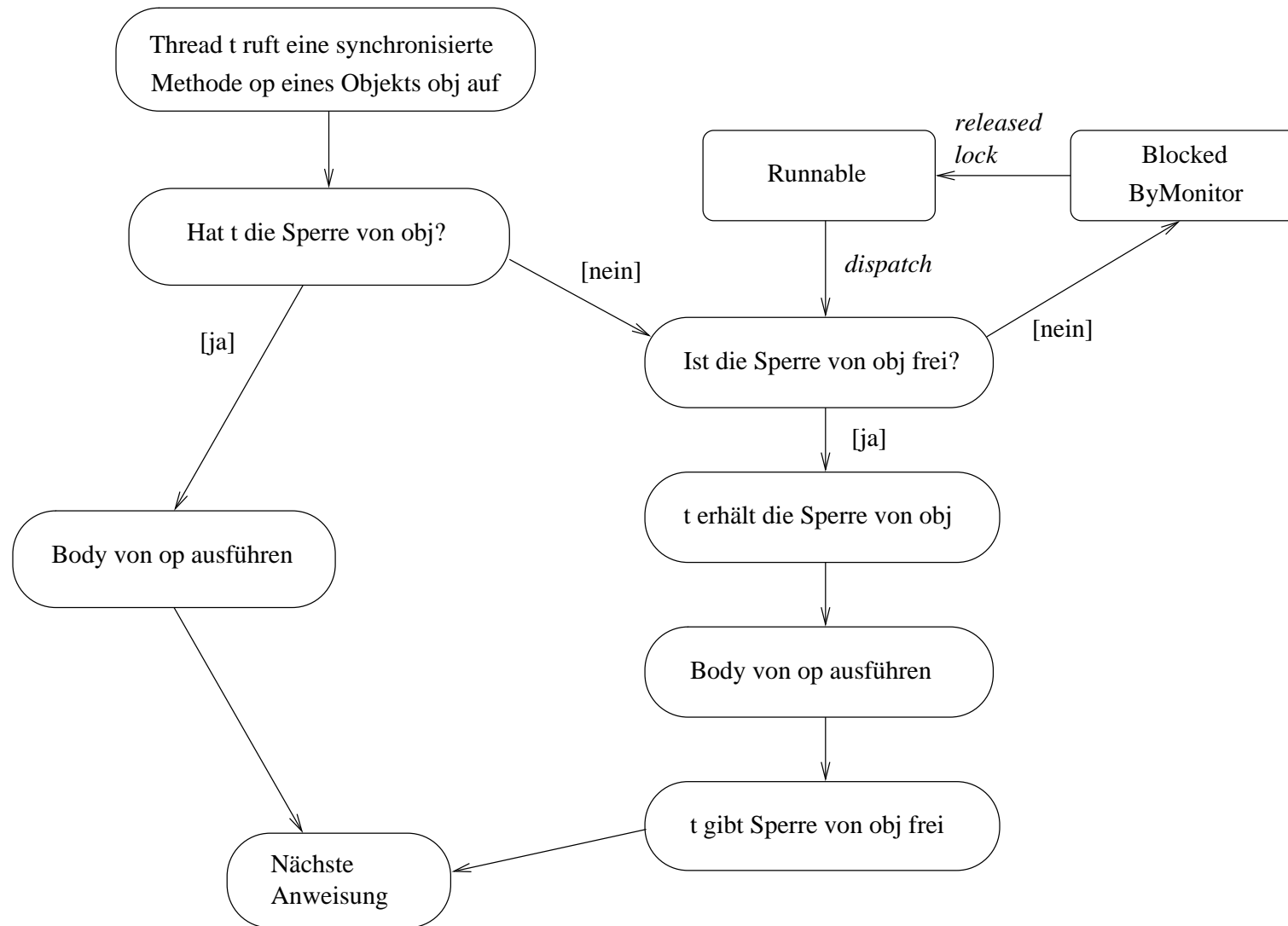
Der wechselseitige Ausschluss in Java wird durch synchronisierte Methoden implementiert:

```
class Monitor {  
    private Data data;  
    synchronized void op1() {...}  
    synchronized void op2() {...}  
}
```

### Wirkungsweise:

Zu jedem Java-Objekt `obj` gibt es eine Sperre ("lock"). Ruft ein Thread `t` eine synchronisierte Methode von `obj` auf, dann bewirbt sich `t` um die Sperre von `obj`. Ist die Sperre bereits (anderweitig) vergeben, so wird `t` blockiert bis die Sperre wieder frei ist. Andernfalls erhält `t` die Sperre und die Methode wird ausgeführt. Mit Beendigung der Methode wird die Sperre zurückgegeben.

## Synchronisation mit Monitoren in Java



## Bemerkungen:

1. Es können auch nur einzelne Blöcke innerhalb eines Methodenrumpfs synchronisiert werden durch:

```
synchronized (obj) {Block}
```

Der Block kann nur dann von einem Thread `t` ausgeführt werden, wenn `t` die Sperre von `obj` erhalten hat. Hierbei ist `void op() {synchronized (this) {Body}}` semantisch äquivalent zu `synchronized void op() {Body}`.

2. Sind nicht alle Methoden einer Klasse synchronisiert, dann kann eine nicht synchronisierte Methode gleichzeitig zu einer synchronisierten Methode ausgeführt werden, z.B.

```
class MyClass {  
    private int i;  
    private double d;  
    int getI() {return i;}  
    synchronized void setD(double x) {d=x;}  
    synchronized double getD() {return d;}  
}
```

3. Beim Überschreiben synchronisierter Methoden in Subklassen muss "synchronized" erneut angegeben werden.