

Kapitel 4

Monitore und wechselseitiger Ausschluss

Prof. Dr. Rolf Hennicker

08.06.2017

4.1 Interferenzen

Parallel ablaufende Prozesse können sich gegenseitig (störend) beeinflussen.

Beispiel (Client/Server):

\parallel TWOCLIENTS_SERVER = (a:CLIENT \parallel b:CLIENT \parallel a:SERVER \parallel 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

Es kommt zu einer Interferenz, da b.service vor a.reply ausgeführt wird.

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

nicht atomar ist

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.

$$\left. \begin{array}{l} ACC = X \\ ACC = ACC + 1 \\ X = ACC \end{array} \right\} \text{realisiert den Service } X = X + 1$$

“service“ ist nicht atomar!

Aktionsfolge 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

*unsynchronisierte
Interferenz*

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.

Monitor
⊖state:Data
+op1() { guarded }
+op2() { guarded }

Beispiel:

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

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

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

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

$a: OP1 = (op1.call \xrightarrow{a.} acquire \xrightarrow{a.} op1.body \xrightarrow{a.} release \xrightarrow{a.} op1.returns \rightarrow OP1).$
 $c: OP2 = (op2.call \xrightarrow{c.} acquire \xrightarrow{c.} op2.body \xrightarrow{c.} release \xrightarrow{c.} op2.returns \rightarrow OP2).$

geschützte Bereiche

Um den wechselseitigen Ausschluss der Operationsrumpfe zu garantieren, verwenden wir eine Sperre:

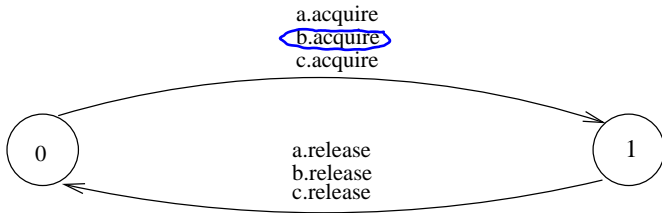
$LOCK = (acquire \rightarrow release \rightarrow LOCK).$

Insgesamt:

$\parallel SYS = (A \parallel B \parallel C \parallel a:OP1 \parallel b:OP1 \parallel c:OP2 \parallel \{a,b,c\}::LOCK).$

3 Threads

*aktuelle Ausführung
des Monitors*



LTS von $\{a,b,c\}::\text{LOCK}$

Mögliche Aktionsfolge:

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

} *einzigste Möglichkeit*

4.3 Wechselseitiger Ausschluss in Java

Der wechselseitige Ausschluss in Java wird durch synchronisierte Methoden implementiert:

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

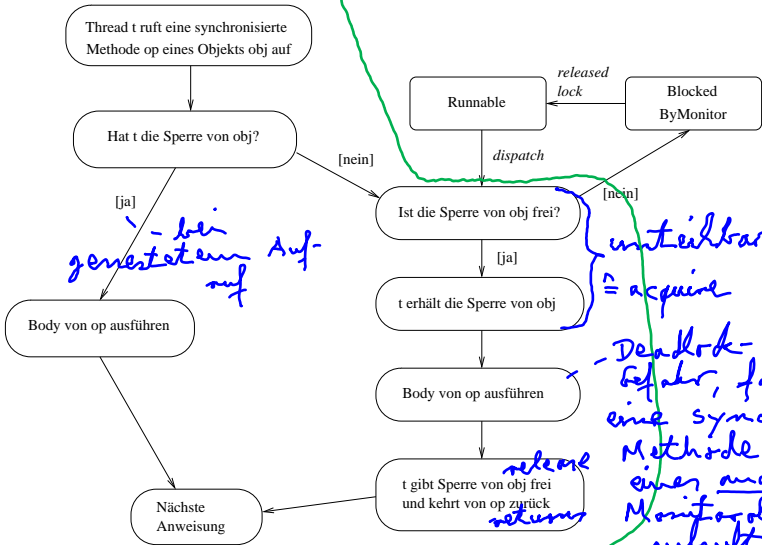
*hier. op2();
müchte
Anweisung*

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

Running



Bspl. II

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 = 1;  
    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.