Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf
Basiert auf ALP IV, SS 2003
Klaus-Peter Löhr
Freie Universität Berlin

Überblick

Überblick

- Interaktion über Objekte
  - Sperrsynchronisation
  - Kritische Abschnitte
  - Monitore
  - Nachrüsten von Ausschluss
  - Leser/Schreiber-Ausschluß
  - Sperroperationen

3 Interaktion über Objekte

3.1 Sperrsynchronisation

- dient der Vermeidung unkontrollierter nebenläufiger Zugriffe auf gemeinsame Datenobjekte und der damit potentiell verbundenen Schmutzeffekte (1.2←)
- Zur Erinnerung:
  wenn nur gelesen wird, droht keine Gefahr (Beispiel: Zeichenketten in Java sind *immutable objects*)
- Gefahr droht, wenn mindestens einer der beteiligten Prozesse, das Datenobjekt *modifiziert*.

3.1 Sperrsynchronisation

- Nachteile von ⟨ ⟩ (1.2.2←):
  - i.a. nicht effizient implementierbar,
  - meist unnötig restriktiv. Beispiel:

```
co ... ⟨ a++; ⟩ ...
 || ... ⟨ a--; ⟩ ...
 || ... ⟨ b++; ⟩ ...
 || ... ⟨ b--; ⟩ ...
 oc
```

  verhindert überlappende Manipulation von a und b – ohne Not!

3.1.1 Kritische Abschnitte

- Syntax in Java
  Java Statement = ... | SynchronizedStatement
  SynchronizedStatement =
  synchronized (Expression) Block
- Der angegebene Ausdruck muß ein Objekt bezeichnen.
- Der angegebene Block heißt auch kritischer Abschnitt (critical section).

3.1.1 Kritische Abschnitte

- Semantik:
  Zwischen kritischen Abschnitten, die sich auf das gleiche Objekt (nicht null) beziehen, ist wechselseitiger Ausschluss (mutual exclusion) garantiert.
- Die zu diesem Zweck praktizierte Synchronisation heisst
  - Sperrsynchronisation (locking) oder
  - Ausschlusssynchronisation (exclusion synchronization)

3.1.1 Kritische Abschnitte

- Beispiel 1: sechs Prozesse:
  - P1: ... synchronized (x) { a++; } ...
  - P2: ... synchronized (x) { a--; } ...
  - P3: ... synchronized (y) { b++; } ...
  - P4: ... synchronized (y) { b--; } ...
  - P5: ... synchronized (z) { big = ... } ...
  - P6: ... synchronized (z) { ... = big; } ...
- mit einer gemeinsamen Variablen double big

3.1.1 Kritische Abschnitte

- Beispiel 2: Ausgabe zusammenhängender Listen
- Spezifikation:

```
interface Printing {// on display (System.out)
boolean request();
  // reserves display, if possible
boolean printLine(String line);
  // prints line, if display reserved for me
boolean release();
  // releases display, if reserved for me
}
```

- Benutzung:

```
if(p.request()) do ... p.printLine(l); ...
.. p.release(); ....
```


3.1.1 Kritische Abschnitte

- Implementierung:

```
class Printer implements Printing {
      .....
private final    String name;
private volatile Thread owner = null;

public boolean request() {
      synchronized(name) {     // or (this) !
        if  (owner==null) {
            owner = Thread.currentThread();
            return true; }
        else return false;
      }
}
```


3.1.1 Kritische Abschnitte

```
public boolean printLine(String line) {
      if  (owner==Thread.currentThread()) {
          System.out.println(line);
      return true; }
    else return false;
}
public boolean release() {
      if  (owner==Thread.currentThread()) {
          owner = null;
          return true; }
      else return false;
}

} // end of class Printer
```

3.1.1 Kritische Abschnitte

- Beachte:
  - Wird ein kritischer Abschnitt vorzeitig beendet (durch Sprung verlassen oder durch Ausnahme abgebrochen), so wird der Ausschluss ordnungsgemäß aufgegeben.
  - Siehe oben return in request.
  - Statische und dynamische Schachtelung von kritischen Abschnitten ist möglich.
  - Befindet sich ein Thread in einem auf das Objekt x bezogenen kritischen Abschnitt, so wird er am Eintritt in einen weiteren auf x bezogenen kritischen Abschnitt nicht gehindert.

3.1.1 Kritische Abschnitte

- Beachte mit Bezug auf das Speichermodell von Java:
  - Beim Eintritt in einen kritischen Abschnitt und beim Verlassen eines kritischen Abschnitts wird der Speicher in einen konsistenten Zustand gebracht.
  - Mit anderen Worten: wenn gemeinsame Variable ausschließlich innerhalb kritischer Abschnitte manipuliert werden, tauchen die in 2.2.3 erwähnten Sichtbarkeits- und Reihenfolgeprobleme nicht auf;
  - volatile im obigen Beispiel 2 ist entbehrlich.

3.1.1 Kritische Abschnitte

- Beachte:
  Verklemmungsgefahr bei dynamisch geschachteltem Sperren:

```
synchronized(x){          synchronized(y){
  ...                       ...
  synchronized(y){          synchronized(x){
    ...                       ...
  }                         }
  ...                       ...
}                         }
```

- ➔ Nichtdeterministische Verklemmung genau dann, wenn x und y auf verschiedene Objekte verweisen.

3.1.1 Kritische Abschnitte

- Alternative Terminologie:

  „Jedes Objekt hat ein Schloss (*lock*), das anfangs offen ist. Ist es offen, ist der Eintritt in den kritischen Abschnitt möglich. Beim Eintritt wird das Schloss geschlossen (gesperrt, *locked*), beim Austritt wieder geöffnet (*unlocked*)."

- Achtung!
  „Das Objekt wird gesperrt" ist falsche, irreführende Terminologie! Es gibt keinen notwendigen Zusammenhang zwischen dem Objekt und den im kritischen Abschnitt manipulierten Daten.

3.1.1 Kritische Abschnitte

- Präzisierung der Semantik von wechselseitigem Ausschluß:

1. *Keine Überlappung* der Ausführung von kritischen Abschnitten, die auf die gleichen Objekte bezogen sind.
2. *Keine Verklemmung* beim Versuch von Prozessen, „gleichzeitig" in einen kritischen Abschnitt einzutreten.
3. *Keine unnötige Verzögerung* beim Eintritt in einen kritischen Abschnitt.
4. *Kein unbegrenzt häufiges Vordrängeln* von Prozessen, die auf den Eintritt in einen kritischen Abschnitt warten: „faire Behandlung" der wartenden Prozesse
   - wird von Java allerdings nicht garantiert!

3.1.2 Monitore (Dijkstra, Brinch Hansen, Hoare, 1972-1974)

- Def.:
  Ein Monitor ist ein Objekt mit
  - vollständiger Datenabstraktion
  - vollständigem wechselseitigem Ausschluss zwischen allen seinen Operationen
- Ideale Syntax (nicht Java!) :
  - Ersetzung von class durch MONITOR oder SYNCHRONIZED CLASS
  - Vorteil: Sicherheit – Invariante bleibt garantiert erhalten!
  - Nachteil: evtl. unnötig restriktiv

3.1.2 Monitore

- Java praktiziert einen Kompromiss: synchronized ist als Modifier einer Methode in einer Klasse verwendbar:
- Syntax in Java:
  synchronized otherModifiers MethodDeclaration
- Semantik bei Objektmethoden:
  - als wäre der Rumpf ein kritischer Abschnitt mit synchronized(this)
- Semantik bei Klassenmethoden (static):
  - als wäre der Rumpf ein kritischer Abschnitt mit synchronized(className.class)

3.1.2 Monitore

- Beispiel 1: tabellierte Abbildung
- Spezifikation:

```
interface Map<Key,Data> {
        // table of (Key,Data) pairs
void enter(Key k, Data d);
Data lookup(Key k);
void remove(Key k);
int  count();
}
```

- (realistisch: Ausnahmen berücksichtigen!)

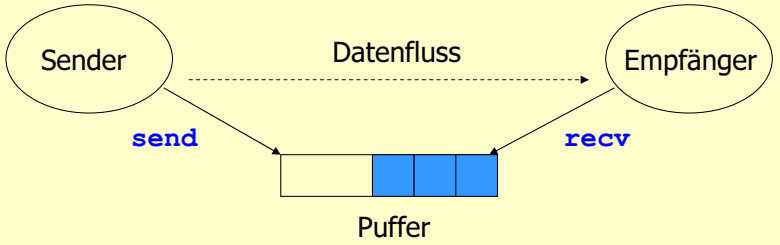
3.1.2 Monitore

- Implementierung:

```
class HashMap<K,D> implements Map<K,D> {
      .....
  public synchronized void enter(K k, D d) {
      .....
  }
  public synchronized D lookup(K k) {
      .....
  }
  public synchronized void remove(K k) {
      .....
  }
  public synchronized int  count() {
      .....
  }
}
```


Beispiel 2:

- Puffer (buffer) =
  Warteschlange (queue)
  von Nachrichten (messages)
  eines Senders (sender, producer)
  an einen Empfänger (receiver, consumer)
  oder auch mehrere Sender/Empfänger



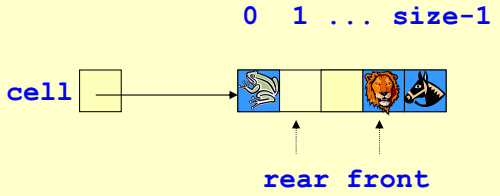
Spezifikation

```
interface Buffer<Message> {
        // like finite-capacity
        //      sequential Queue (ALP III!)

void send(Message m) throws Overflow;
        // like enqueue

Message recv() throws Underflow;
        // like dequeue
}
```


Repräsentation als linearer „Ringpuffer" in Feld:




```
class LinearBuffer<M> implements Buffer<M> {
public LinearBuffer(int size) {
      this.size = size;
      cell = new M[size];}
private int size,count,front,rear;
private M[] cell;  // element container

   // invariant:
   // (front+count)%size == rear  &
   // 0 <= count <= size  & .....
```

Implementierung:

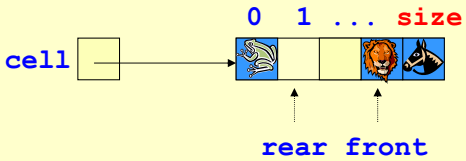
```
public void send(M m) throws Overflow {
      if(count==size) throw new Overflow();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
}

public M recv() throws Underflow{
      if(count==0) throw new Underflow();
      M m = cell[front];
      count--;
      front = (front+1)%size;
     return m;
}
}
```


Implementierung

```
synchronized
public void send(M m) throws Overflow {
      if(count==size) throw new Overflow();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
}
synchronized
public M recv() throws Underflow{
      if(count==0) throw new Underflow();
      M m = cell[front];
      count--;
      front = (front+1)%size;
     return m;
}
}
```


Repräsentation, die mit weniger Ausschluß auskommt ?




```
class LinearBuffer<M> implements Buffer<M> {
public LinearBuffer(int size) {
      this.size = size;
      cell = new M[size+1];}
private int size,front,rear;   // no count !
private M[] cell;  // element container

   // invariant:
   // 0 <= front <= size  &  .....
```


Implementierung:

```
     int count() {
        return (rear-front+size+1)%(size+1);
     }
public void send(M m) throws Overflow {
     synchronized(this) { // sender exclusion
        if(count()==size) throw new Overflow();
        cell[rear] = m;
       rear = (rear+1)%(size+1);}
}
public M recv() throws Underflow{
     synchronized(cell) { // receiver exclusion
        if(count()==0) throw new Underflow();
        M m = cell[front];
       front = (front+1)%(size+1);
       return m; }
}
}
```

Bemerkungen

1. send und recv operieren stets auf verschiedenen Zellen des Feldes cell – dort kann es also keinen Konflikt zwischen send und recv geben, der zu Nichtdeterminismus führen würde.
2. send-Inkarnation werden gegeneinander ausgeschlossen, ebenso recv-Inkarnationen (wobei die Wahl der Objekte this und cell hier relativ willkürlich ist).
3. Die Tatsache, dass count nicht unteilbar ist, ist unproblematisch. Auch kann count jederzeit von anderen Stellen des Programms her aufgerufen werden.

Achtung:

- Die Lösung ist dennoch *nicht korrekt,*
  - weil beispielsweise bei leerem Puffer ein recv die Effekte der beiden Zuweisungen eines send in verkehrter Reihenfolge beobachten kann und damit aus einer Zelle liest, bevor sie beschrieben wird.
  - Dieses Problem ist Java-spezifisch und tritt bei anderen Sprachen nicht auf.
- Die Verwendung von volatile würde hier nicht helfen, da es nicht auf die Feldelemente bezogen werden kann.

... und drei Übungsfragen:

1. Warum kann die Prüfung auf Über/Unterlauf nicht aus den kritischen Abschnitten herausgezogen werden?
2. Kann return m aus dem kritischen Abschnitt herausgezogen werden?
3. Wie muß eine zusätzliche Operation urgent aussehen, die eine dringliche Nachricht vorn im Puffer ablegt?

3.1.3 Nachrüsten von Ausschluss...

- ... bei Klassen, die für *sequentielle* Benutzung gebaut wurden,
- für Objekte, die *nichtsequentiell* benutzt werden,
- durch Bereitstellung eines Monitors, der als synchronisierte („*thread-safe*“) Version der Klasse eingesetzt werden kann,
- typischerweise mit *gleicher Schnittstelle*.

2 Alternativen:

1. *Vererbung:* Monitor ist *Unterklasse* der Originalklasse
2. *Delegation:* Monitor ist *Adapter* für die Originalklasse

- Beispiel:

```
interface SortedSet { ...
  boolean add     (Object x);
  boolean contains(Object x);
}
class TreeSet implements SortedSet { ...
  public boolean add     (Object x) {.....}
  public boolean contains(Object x) {.....}
}
```

voll synchronisiert, da die Implementierung des Originals i.a. unbekannt ist !

Vererbung und Delegation

1.

```
class SyncTreeSet extends TreeSet {...
  public synchronized boolean add(Object x){
                        return super.add(x); }
  public synchronized boolean contains(Object x){
                    return super.contains(x); }
}
```

2.

```
class SyncTreeSet implements SortedSet {...
  private final SortedSet s = new TreeSet();
  public synchronized boolean add(Object x){
                        return s.add(x);     }
  public synchronized boolean contains(Object x){
                    return s.contains(x);     }
}
```


Fabriken (Factories)

- Synchronisierte Adapter kann man über die *Fabrik* java.util.Collections erhalten, z.B. mittels

```
public static SortedSet synchronizedSortedSet
                          (SortedSet x) {
    return new SortedSet() {
      final SortedSet s = x;
      ...
    }
}
```

anonyme Klasse analog zu SyncTreeSet

- oder

```
public static List synchronizedList(List x) {
    return new List() {...}
}
```

- usw.

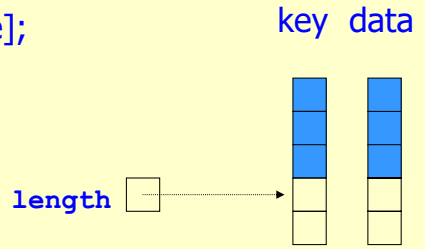
Java-Bibliotheken

- Beachte: Java-Bibliotheksklassen
  - sind teilweise synchronisiert (z.B. Vector),
  - *zum großen Teil aber nicht !*
  - *Insbesondere ist* clone *nicht synchronisiert !*

3.1.4 Leser/Schreiber-Ausschluß

- (reader/writer exclusion)
- Beispiel: (vgl. Map, HashMap, 3.1.2←)

```
class LinearMap<K,D> implements Map<K,D> {
public  LinearMap() {.....}
private int size;
private int length = 0;
private K[] key = new K[size];
private D[] data= new D[size];
```



Leser/Schreiber-Ausschluß

```
public synchronized void enter(K k, D d) {
    for (int i=0; i<length; i++)
        if (k==key[i])
            {data[i] = d; return;}
    if (length<size) {
        key[length] = k; data[length]= d;
        length++; }
    return;
}

public synchronized void remove(K k) {
    .....
}
```


Leser/Schreiber-Ausschluß

```
public synchronized D lookup(K k) {
    for(int i=0; i<length; i++)
        if(k==key[i]) return data[i];
    return null;
}

public synchronized int count() {
    return length;
}
```

- Ausschluß ist unnötig strikt: lookup, count lesen nur !
- ➔ count braucht nicht **synchronized** zu sein
- ➔ lookups könnten einander überlappen

Leser/Schreiber-Ausschluß

- Wünschenswertes Sprachkonstrukt anstelle von synchronized :
  - Statement = ... | **READING**(Expression) Block
    | **WRITING**(Expression) Block
- *Semantik:*
  Wie bei synchronized
  - wechselseitiger Ausschluss von Abschnitten, deren Expression sich auf das gleiche Objekt bezieht –
  - außer wenn beide READING sind
- Statische oder dynamische Schachtelung möglich, z.B. auch *upgrading:*
  READING(x){... **WRITING**(x){...} ... }

Leser/Schreiber-Ausschluß

- ➔

```
public WRITING void enter(K k, D d) {
      .....
}
public WRITING void remove(K k) {
      .....
}
public READING D lookup(K k) {
      .....
}
public int count() {
      .....
}
```


Leser/Schreiber-Ausschluß

- Achtung:
- Die wünschenswerte Fairness (3.1.1←) ist u.U. *nicht garantiert:*
  - Wenn eine nicht abbrechende Reihe von Lesern sich „die Klinke in die Hand gibt", kommen Schreiber nicht zum Zuge.

3.1.5 Sperroperationen

- Alternative zum Sperren mittels syntaktischer Klammerung:

Sperroperation;
kritischer Abschnitt;
Entsperroperation;

spezielle *Anweisungen* zum Sperren/Entsperren

- Motivation?

Motivationen

1. *Sequentielle Sprache:* kennt keine Synchronisationskonstrukte ➔ Einsatz von *Bibliotheksroutinen*
2. *Übersetzerbau:* Letztlich müssen auch klammernde Konstrukte in *Anweisungen* umgesetzt werden.
3. *Restriktionen: reine Schachtelung* von kritischen Abschnitten kann u.U. zu restriktiv sein.

Sperroperationen

- (Nicht Java!)
- void lock()
  - *„setzt Sperre*"
  - sperrt einen objektbezogenen kritischen Abschnitt, gegebenenfalls nach Warten auf Freigabe durch einen anderen Prozess
- void unlock()
  - *„löst Sperre*"
  - gibt einen zuvor gesperrten kritischen Abschnitt frei
- synchronized(x){...}
  - ist fast äquivalent zu x.lock();...; x.unlock();

Sperroperationen

- Gefahr des Vergessens von unlock bedeutet *unsicheres Programmieren !*
- *Beispiele:*
  - lock();... **return**;... unlock();...
  - lock(); **if** (C) {... unlock();...} a = b; ...
  - **try** {lock(); .... unlock();...} **catch**(Exception e) {...}

Vorteile

- Sperren und Entsperren in *verschiedenen* Routinen möglich
- Zwang zur Schachtelung entfällt (Verklemmungsgefahr bleibt!)
- Beispiel:

```
tape.lock();              ...
printer.lock();           ...
 ...                      printer.lock();
 ...                      ...
tape.unlock();            ...
 ...                      tape.lock();
printer.unlock();         ...
```


Lese/Schreibsperren

- Sperroperationen für den Leser/Schreiber-Ausschluss:
  - Rlock() „setzt Lesesperre"
  - Wlock() „setzt Schreibsperre"
  - unlock() „löst (gibt frei) Sperre"
- Erlaubt ist auch *upgrading* (3.1.3←) einer Sperre :
  - Rlock();... Wlock();... unlock();

3.1.6 Sperrsynchronisation in Datenbanken

- Datenbank *(database)*
  enthält große Menge von langzeitgespeicherten Daten

- Transaktion *(transaction)*
  besteht aus mehreren lesen/schreibenden Einzelzugriffen auf Daten der Datenbank:

```
BEGIN  ...
   ...        → ABORT   Abbruch möglich
   ...
COMMIT
```


Sperrsynchronisation in Datenbanken

- Datenbank als *Monitor*, d.h. keine überlappenden Transaktionen ?
  - wäre viel zu restriktiv

- → möglichst *sparsam sperren*, nur dort, wo wirklich auf die gleichen Daten zugegriffen wird

- Aber so, daß Korrektheit gewährleistet ist !

Sperrsynchronisation in Datenbanken

Datenbank	**komplexes Objekt**
Transaktion	Operation
Konsistenz *(consistency)*	Invariante
Isolation *(isolation)*	Serialisierbarkeit
Atomarität *(atomicity)*	bei Abbruch Leeroperation
Langzeitspeicherung *(durability)*	Persistenz

- „ACID – Eigenschaften"

Serialisierbarkeit

- Def.: Ein Ablauf nebenläufiger Operationen heißt serialisierbar, wenn er den gleichen Effekt und die gleichen Ergebnisse hat, als würden die Operationen *in irgendeiner Reihenfolge streng nacheinander* ausgeführt.
- Wünschenswert ist, daß alle Abläufe serialisierbar sind !

- Def.: Korrektheitskriterium für nebenläufig benutzte Objekte:
  - Ein Objekt heißt serialisierbar, wenn jeder mögliche Ablauf serialisierbar ist.
  - (→3.1.7)

Ein Beispiel

```
class Account {
private long bal; ...

public long balance() {
      Rlock();
        long b = bal;
     unlock();
      return b;
}
public void deposit(long a) {
      Wlock();
        bal += a;
      unlock();
     return;
}
public void withdraw(long a) {
      ... }
```


Ein Beispiel/2

```
public void transfer(long a, Account d) {
      Wlock();
        bal -= a;
      unlock();
      d.Wlock();
        d.bal += a;
      d.unlock();
      return;
}
```


Ein Beispiel/3

```
public static long total(Customer c) {
      // sum of balances of customer's
      // checking and savings accounts
      long sum = 0;
     c.c.Rlock();
        sum += c.c.bal;
      c.c.unlock();
      c.s.Rlock();
        sum += c.s.bal;
      c.s.unlock();
     return sum;
}

}
```


Ein Beispiel-Ablauf

<table>
<tr><td colspan="2">{c.c.bal == 70, c.s.bal == 30}</td></tr>
<tr><td>total(c)</td><td>c.c.transfer(60,c.s)
c.c.Wlock();
c.c.bal -= a;
c.c.unlock();</td></tr>
<tr><td colspan="2">{c.c.bal == 10}</td></tr>
<tr><td>c.c.Rlock();
sum += c.c.bal;
c.c.unlock();</td><td></td></tr>
<tr><td colspan="2">{sum == 10}</td></tr>
<tr><td>c.s.Rlock();
sum += c.s.bal;
c.s.unlock();</td><td></td></tr>
<tr><td colspan="2">{sum == 40} statt 100 !</td></tr>
<tr><td></td><td>c.s.Wlock();
c.s.bal += a;
c.s.unlock();</td></tr>
<tr><td colspan="2">{c.s.bal == 90} ok</td></tr>
</table>

Serialisierbar?

- Der gezeigte Ablauf ist nicht serialisierbar, weil *unzureichend gesperrt wird !*
- ➔ Überlappung von total und transfer unterbinden ?
  - **static** Lock x einführen und
  - total zusätzlich mit x.Rlock() und
  - transfer zusätzlich mit x.Wlock() sperren
- ➔ Damit wird wiederum *zu strikt gesperrt:*
  Ein Leser/Schreiber-Ausschluß erfolgt auch bei Operationen auf gänzlich verschiedenen Konten.

Idee für problemadäquate Lösung

- Mit „Mehrobjekt-Sperroperation" lock(ob1,ob2,...) so programmieren:

```
public void transfer(long a, Account d) {
    Wlock(this,d);
      bal   -= a;
      d.bal += a;
    unlock(this,d);
    return;
}
public static long total(Customer c) {
    long sum = 0;
    Rlock(c.c,c.s);
      sum += c.c.bal;
      sum += c.s.bal;
    unlock(c.c,c.s);
    return sum;
}
```


Zwei-Phasen-Sperren

- Diese Idee ist Grundlage des Zwei-Phasen-Sperrens *(two-phase locking, 2PL)* bei Datenbanken
- 4 Varianten:
  1. *konservativ und strikt:*
     alle benötigten Sperren am Anfang anfordern und am Schluss freigeben
  2. *konservativ:*
     alle benötigten Sperren am Anfanganfordern, aber *baldmöglichst freigeben*
  3. *strikt:*
     jede Sperre *spätmöglichst anfordern* und alle am Schluss freigeben
  4. *(sonst:)*
     jede Sperre *spätmöglichst anfordern* und *baldmöglichst freigeben*
- *„2 Phasen": erst Sperren setzen, dann Sperren lösen.*

Zwei-Phasen-Sperren

- Bei Datenbanken ist i.a. nur 3) praktikabel, weil das Sperren implizit durch das Datenbanksystem erledigt werden soll, d.h. nicht Teil des Anwendungscodes ist.
- Bei explizitem Sperren ist 4) am günstigsten, da am wenigsten einschränkend für die Nichtsequentialität.
- Beachte:
  - Bei 3) und 4) drohen *Verklemmungen*!
  - Bei 1) und 2) ist *keine Fairness* garantiert!

Zwei-Phasen-Sperren

konservativ & strikt	konservativ	strikt	-

transfer
Wlock(this,d);
bal -= a;
 	
d.bal += a;
unlock(this,d);
	Wlock(this,d);
	bal -= a;
	unlock(this);
	
	d.bal += a;
	unlock(d);
		Wlock(this);
		bal -= a;
		Wlock(d);
		
		d.bal += a;
		unlock(this,d);
			Wlock(this);
			bal -= a;
			Wlock(d);
			unlock(this);
			d.bal += a;
			unlock(d);

total
Rlock(c.c,c.s);
sum += c.c.bal;
 	
sum += c.s.bal;
unlock(c.c,c.s);
	Rlock(c.c,c.s);
	sum += c.c.bal;
	unlock(c.c);
	
	sum += c.s.bal;
	unlock(c.s);
		Rlock(c.c);
		sum += c.c.bal;
		Rlock(c.s);
		
		sum += c.s.bal;
		unlock(c.c,c.s);
			Rlock(c.c);
			sum += c.c.bal;
			Rlock(c.s);
			unlock(c.c);
			sum += c.s.bal;
			unlock(c.s);

Zwei-Phasen-Sperren

konservativ & strikt	konservativ	strikt	-
deposit Wlock(); bal += a; unlock()			

balance
Rlock();
long b = bal;
unlock();			

Zusammenfassung

Zusammenfassung

- Interaktion über Objekte
  - Sperrsynchronisation
  - Kritische Abschnitte
  - Monitore
  - Nachrüsten von Ausschluss
  - Leser/Schreiber-Ausschluß
  - Sperroperationen