Nebenläufige Programmierung II

Martin Wirsing

in Zusammenarbeit mit
Matthias Hölzl, Piotr Kosiuczenko, Dirk Pattinson

06/03

Ziele

- Verklemmungsprobleme als mögliche Folge von Synchronisation verstehen lernen
- Kommunikation zwischen Threads durch aktive und passive Benachrichtigung verstehen lernen
- Standardbeispiele für nebenläufige Programme wie z,B. das Erzeuger/Verbraucherproblem kennen lernen

Verklemmung („Deadlock")

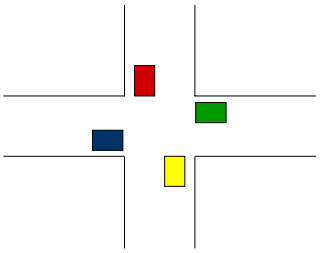
Durch Synchronisation können sich aber **Lebendigkeitsprobleme** ergeben:
Das Programm kann in eine Verklemmung geraten.

Man spricht von **Verklemmung**, wenn es mindestens einen Thread eines Programms gibt, der nicht beendet ist, aber nicht weiterarbeiten kann, da die von ihm benötigten Ressourcen nicht freigegeben sind.

Typischerweise benötigt man zu einer Verklemmung mindestens zwei Threads.

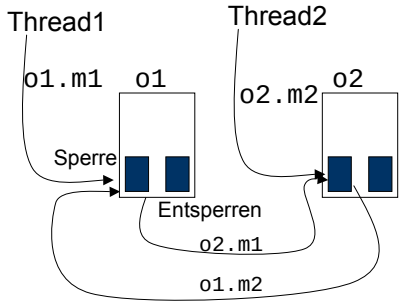
Verklemmung

Beispiel: Straßenkreuzung mit Rechts-vor-Links-Regel, bei der an jeder Straßeneinfahrt ein Auto steht:



Jedes Auto möchte die Straßenkreuzung überqueren, kann aber nicht, da es die Vorfahrt von rechts respektieren muß. Dadurch sind alle 4 Autos blockiert.

Verklemmung in der Programmierung



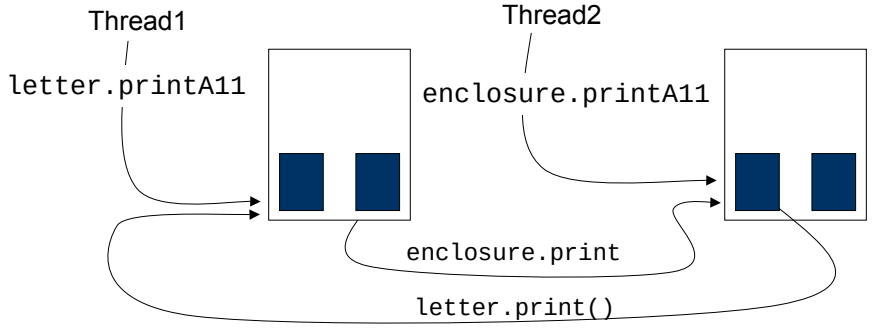
1. Thread 1 beginnt die Ausführung der synchronisierten Methode `m1` bezüglich `o1`.
2. Thread 2 beginnt die Ausführung der synchronisierten Methode `m2` bezüglich `o2`.
3. Während der Ausführung von `o1.m1` wird von Thread 1 versucht mit der synchronisierten Methode `m1` das gesperrte Objekt `o2` aufzurufen.
4. Während der Ausführung von `o2.m2` wird von Thread 2 versucht mit der synchronisierten Methode `m2` das gesperrte Objekt `o1` aufzurufen.

Damit warten Thread 1 und Thread 2 darauf, daß jeweils der andere das gesperrte Objekt verläßt ⇒ **Verklemmung!**

Verklemmung in der Programmierung

Beispiel: Eine Dokumentenklasse, bei der Dokumente aus zwei Teilen bestehen, z.B. aus einem Brief und einer Anlage.

Falls nun `letter` ein Dokument mit Anlage `enclosure` und `enclosure` ein Dokument mit Anlage `letter` ist, Verklemmung ergibt sich beim Drucken dieser Dokumente mit Hilfe 2er verschiedener Threads z.B. :



Print

```
public class Document
{       Document otherPart;
        public synchronized void print()
        {       System.out.println(„first line");
                //...
                 System.out.println(„last line");
        }
        public synchronized void printAll()
        {       otherPart.print();
                print();
        }
}
```


Verklemmung in der Programmierung

Es gibt keine allgemeine Methode zur Beseitigung von Verklemmungen. Die Lebendigkeit eines Programms muß durch einen guten Entwurf gesichert werden.

Mögliche Vorgehensweisen sind folgende:

1. **Top-Down-Methode** (Sicherheit zuerst):
   Entwurf von Methoden und Klassen mit voller Synchronisation und dann schrittweises Aufheben unnötiger Synchronisation.

2. **Bottom-Up-Methode** (Lebendigkeit zuerst):
   Entwurf der Methoden und Klassen ohne Synchronisation und dann nachträgliches Hinzufügen von Synchronisation durch spezielle Techniken.

Kommunikation zwischen Threads

Synchronisation genügt nicht für die Kommunikation zwischen Threads.

Wie das folgende Beispiel zeigt braucht man zusätzliche Mechanismen der Benachrichtigung.

Erzeuger/Verbraucher Problem

Beim Erzeuger/Verbraucherproblem gibt es einen Puffer, einen Erzeuger und einen Verbraucher.

Der Erzeuger produziert unaufhörlich Dinge (hier Zahlen) und schreibt sie in den Puffer; der Verbraucher liest unaufhörlich Elemente aus dem Puffer.

Erzeuger

```
//Die Klasse Producer schreibt der Reihe nach die Zahlen 0,1,2,... in
//einenPuffer.
public class Producer implements Runnable
{       private AbstractBuffer q;
/**
Der Konstruktor initialisiert einen Puffer und erzeugt einen Thread.
*/
        public Producer(AbstractBuffer q)
        {       this.q = q;
                new Thread(this, "Producer").start();
        }
Diese Methode schreibt der Reihe nach die Zahlen 0,1,2,... in den
aktuellen Puffer.
public void run()
        {       int i = 0;
                while(true)
                {       q.put(i++);
                }
        }
}
```


Verbraucher

```
/**
Diese Klasse liest mit Hilfe eines Threads unaufhörlich Werte aus einem
Puffer.
*/
public class Consumer implements Runnable
{       AbstractBuffer q;
        public Consumer(AbstractBuffer q)
        {       this.q = q;
                new Thread(this, "Consumer").start();
        }

        public void run()
        {       while(true)
                {  q.get();
                }
        }
}
```

Schnittstelle für einelementigen Puffer

```
public interface AbstractBuffer
{
/**
Zugriff auf das Element im Puffer
@return         Wert des Puffers
*/
        public  int get();

 /**
Überschreiben des Elements im Puffer mit neuem Wert
@param m        neuer Wert des Puffers
*/
        public  void put (int m);

}
```


Beispiel: Einelementiger Puffer für ganze Zahlen

```
public class SimpleBuffer implements AbstractBuffer
{       private int element;
        public SimpleBuffer (int m)
         { element = m;
         }
        //Zugriff auf das Element im Puffer
        public synchronized int get()
         {       System.out.println("Got: " + element);
                 return element;
         }
        //Überschreiben des Elements im Puffer mit neuem Wert
         public synchronized void put (int m)
         {       element = m;
                 System.out.println("Put: " + m);
         }
}
```

Strategien zur Benachrichtigung von Threads

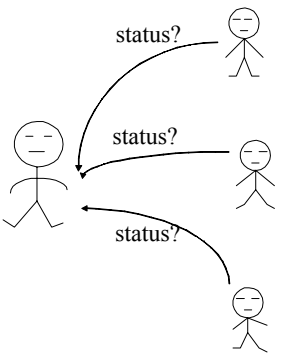
Communication between threads is often based on **notifications**:

- **passive**: someone asks me if I have changed
- **timer-based**: someone wakes me up
- **active**: an observable notifies one or all observers

Aktives Warten (Polling)

Active Waiting (Polling)

Strategy:
An object repeatedly checks for a conditon until is true, and then reacts to it.



observer

Passive object (observable)

(see Coad 97)

Aktives Warten (Polling)

Schema ...

```
while (!conditionChanged)
        ;
actOnIt;
```


Beispiel für Aktives Warten (Polling)

```
public Vector hasMoreRoom (double maxprice)
{ Vector m = new Vector();
  while(!reservations_.size()<description_.getBookableSeats().size())
        ;
  //(*)
  // get all seats with price ≤ maxprice
  return m;
}
```

Problems:

- Resource and time consuming
- Can be unsafe (if another reservation is made at *). Note that trying to acquire a lock is a kind of (safe) active waiting.

Polling aus einelementigem Puffer

```
/**
Implementierung eines einelementigen Puffers durch Polling
*/
public class PollBuffer implements AbstractBuffer
{
        private int element;
        private boolean valueSet = false;
//Konstruktor
         public PollBuffer (int m)
         {       element = m;
                 valueSet = false;
         }
```


Polling aus einelementigem Puffer

```
Zugriff auf das Element im Puffer
public int get()
        {       while (!valueSet) //Polling, bis valueset == true
                        ;
                System.out.println("Got: " +  element);
                valueSet = false;
                return element;
                }

// Überschreiben des Elements im Puffer mit neuem Wert
public  void put(int n)
        {       while (valueSet) //Polling, bis valueset == false
                        ;
                        element = n;
                        valueSet = true;
                        System.out.println("Put: " + n);
        }
```

Aktive Benachrichtigung (Active Notification)

Active notification puts the notification responsibility within the object that changes. The observer waits until it is notified.

Java provides two methods from class Object:

- **o.wait()**
  suspends the invoicing thread, releases the object lock

  (both in one step) and places it on o's waiting list for notifications
- **o.notifyAll()** (**o.notify()**)

  reactivates **all** (**one**, resp.) of the threads in the notification list.

Aktive Benachrichtigung (Active Notification)

Note:

- *synchronized:* `wait`, `notify` and `notifyAll` can only be called within synchronized blocks.
- *anonymous:* a notifying thread can't specify which thread(s) it will reactivate.
- *contentless:* a reactivated thread is not informed what happened to the trigger of the notification.
- *not guaranteed to be fair:* there is no required order in which to notify threads. (However, fairness is desirable for virtual machine implementations).

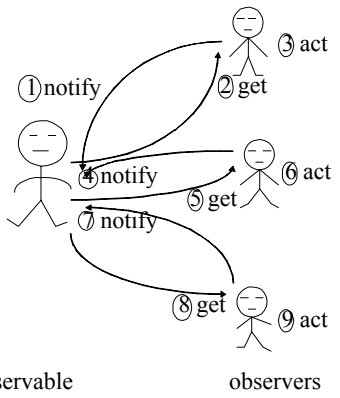
Aktive Benachrichtigung (Active Notification)

Observer:

waits for notification of changed condition.

Observable:

notifies all (or one) observers of some change of condition.



(cf. Coad 97)

Aktive Benachrichtigung (Active Notification)

Observer Schema:

```
synchronized void doWhenCondition () {
        while (! condition)
                try {wait();}
                catch (InterruptedException e) {...}
        // Do what is needed when condition is true
}
```

Observable Schema:

```
synchronized void changeCondition () {
        // ... change some value used in the condition test
        notifyAll();
}
```

Beispiele für Aktive Benachrichtigung

Example:
Readers/Writers of Critical Regions

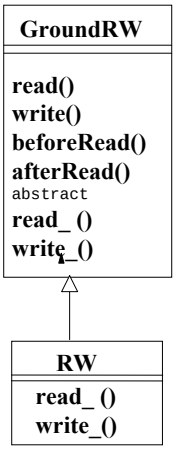
Several readers can concurrently access a critical region, but a writer has exclusive access.

A class `RW` extends the `GroundRW` class which provides abstract und protected `read_()` and `write_()` methods. `RW` allows multiple policies for concurrency control.

Lesen und Schreiben kritischer Regionen

It maintains counts of active and passive readers and writers. The methods `beforeRead()`, `afterRead()`, `beforeWrite()`, `afterWrite()` implement the chosen synchronization policy.

The `beforeRead()` method waits until reading is allowed; `afterRead()` notifies all waiting readers and writers



Lesen und Schreiben kritischer Regionen

```
public abstract class GroundRW {
  protected int activeReaders_ = 0;  // threads executing read_
  protected int activeWriters_ = 0;  // always zero or one
  protected int waitingReaders_ = 0; // threads not yet in read_
  protected int waitingWriters_ = 0; // same for write_

  protected abstract void read_(); // implement in subclasses
  protected abstract void write_();
```


Lesen und Schreiben kritischer Regionen

```
public void read()  {
    beforeRead();
    read_();
    afterRead();
  }

  public void write() {
    beforeWrite();
    write_();
    afterWrite();
  }
```

Lesen und Schreiben kritischer Regionen

```
protected boolean allowReader()
                                    // allowReader defines reading policy
{    return waitingWriters_ == 0 && activeWriters_ == 0;
}
 protected boolean allowWriter() {
   return activeReaders_ == 0 && activeWriters_ == 0;
 }
```


Lesen und Schreiben kritischer Regionen

```
protected synchronized void beforeRead() {
   ++waitingReaders_;
   while (!allowReader())              // wait until reading is allowed
     try { wait(); } catch (InterruptedException ex) {}
   --waitingReaders_;
   ++activeReaders_;
 }

 protected synchronized void afterRead()  {
   --activeReaders_;
   notifyAll();          // notifies waiting readers and writers

 }
```

Lesen und Schreiben kritischer Regionen

```
    protected synchronized void beforeWrite() {
      ++waitingWriters_;
      while (!allowWriter())
        try { wait(); } catch (InterruptedException ex) {}
      --waitingWriters_;
      ++activeWriters_;
    }

    protected synchronized void afterWrite() {
      --activeWriters_;
      notifyAll();
    }
  }
```


Beispiel: Einelementiger Puffer für ganze Zahlen

```
public class Buffer implements AbstractBuffer
{
        private int element;
        private boolean valueSet = false;

//Konstruktor
        public Buffer (int m)
        {       element = m;
                valueSet = false;
        }
```

Beispiel: Einelementiger Puffer für ganze Zahlen

```
/**
Zugriff auf das Element im Puffer
@return         Wert des Puffers
*/
        public synchronized int get()
        {       while (!valueSet)  //Warten bis valueset == true
                {       try {wait();}
                        catch(InterruptedException e){}
                }

        //Hier ist valueset == true: Änderung von valueset und
Benachrichtigung
                System.out.println("Got: "+ element);
                valueSet = false;       // "Anderung der Bedingung
valueSet
                notify();
                return element;
        }
```


Beispiel: Einelementiger Puffer für ganze Zahlen

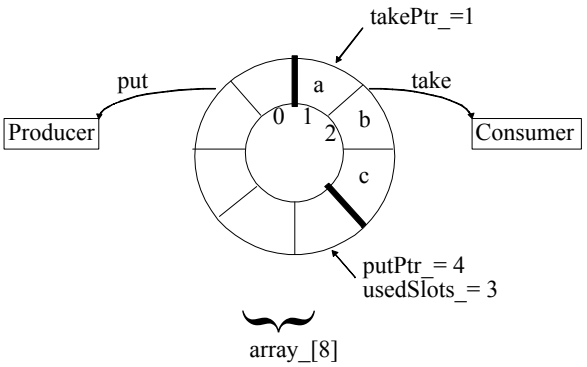
```
/**
Überschreiben des Elements im Puffer mit neuem Wert
@param m        neuer Wert des Puffers
*/
        public synchronized void put(int n)
        {       while (valueSet)        //Warten bis valueset == false
                {       try {wait();}
                        catch(InterruptedException e){}
                }

//Hier ist valueset == false: Änderung von element und valueset und
//Benachrichtigung
                element = n;
                valueSet = true; // "Anderung der Bedingung valueSet
                System.out.println("Put: " + n);
                notify();
        }
    }
```

Erzeuger/Verbraucher allgemein

A producer/consumer system uses a **bounded ring buffer**.

A consumer can **take** an element only **from a non-empty buf-fer** (i.e. usedSlots_ > 0).
A producer can **put an element** on the buffer only if the buffer is **not full** (i.e. usedSlots_ < array_.length).



Ein allgemeiner beschränkter Puffer

```
public class BoundedBufferVST implements BoundedBuffer {
  protected Object[]  array_;      // the elements
  protected int putPtr_ = 0;       // circular indices
  protected int takePtr_ = 0;
  protected int usedSlots_ = 0;    // the count

  public BoundedBufferVST(int capacity)
   throws IllegalArgumentException {
    if (capacity <= 0) throw new IllegalArgumentException();
    array_ = new Object[capacity];
  }

  public int count() { return usedSlots_; }
  public int capacity() { return array_.length; }
```

Ein allgemeiner beschränkter Puffer

```
public synchronized void put(Object x) {
    while (usedSlots_ == array_.length) // wait until not full
      try { wait(); } catch(InterruptedException ex) {};
    array_[putPtr_] = x;
    putPtr_ = (putPtr_ + 1) % array_.length; // cyclically inc
    if (usedSlots_++ == 0)
      notifyAll();
  }
  public synchronized Object take() {
    while (usedSlots_ == 0) // wait until not empty
      try { wait(); } catch(InterruptedException ex) {};
    Object x = array_[takePtr_];
    array_[takePtr_] = null;
    takePtr_ = (takePtr_ + 1) % array_.length;
    if (usedSlots_-- == array_.length)
      notifyAll();
    return x;
  }
}
```


Zusammenfassung

- Java unterstützt Nebenläufigkeit durch „leichtgewichtige“ Prozesse, sogenannte Threads, die über die gemeinsame Halde und damit über gemeinsam benutzte Objekte miteinander kommunizieren.
- Nebenläufigkeit wirft Sicherheits- und Lebendigkeitsprobleme auf. Gemeinsam benutzte Objekte müssen synchronisiert werden. Zuviel Synchronisation kann Verklemmungen hervorrufen.
- Wichtige Kommunikationsstrategien sind „Polling“, d.h. immer wiederkehrende Anfragen, und „Benachrichtigung“. Meist ist die Benachrichtigungsstrategie wegen des geringeren Aufwandes bei der Kommunikation vorzuziehen.