Programmieren II

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2010

Deadlocks

Wait/Notify

Reflections

JavaBeans

Verklemmungen

Solange es nur einen Lock bzw. nur ein Objekt gibt, über das kritische Bereiche synchronisiert werden, tauchen keine Probleme auf.

Gibt es aber z. B. mehrere Locks, so kann es in der Praxis zu *Verklemmungen* bzw. *deadlocks* kommen. Hier blockiert das Programm (oder ein Teil des Programms), wenn es zu einer Situation kommt, wo Threads einen Lock besitzen und darauf warten, dass ein anderer frei wird, dieser jedoch von einem Thread gehalten wird, der wieder darauf wartet, dass ein anderer frei wird, . . .

Das Philosophenproblem

Eine solche Verklemmung wird musterhaft durch das *Philosphenproblem* beschrieben:

- Mehrere Philosophen sitzen an einem runden Tisch und wollen Spaghetti essen.
- Zwischen zwei Philosophen liegt jeweils eine Gabel, die beide sich teilen.
- Zum Essen laufen folgende Schritte ab:
  - Der Philosoph nimmt die linke Gabel in die Hand (ggfls. wartet er, bis sie frei ist),
  - dann nimmt er die rechte Gabel in die Hand (sobald sie frei ist),
  - er isst, bis er satt ist,
  - beide Gabeln werden wieder abgelegt.
- Nach dem Essen wird jeweils eine Denkpause eingelegt, bis der Philosoph wieder hungrig wird.

Schaubild

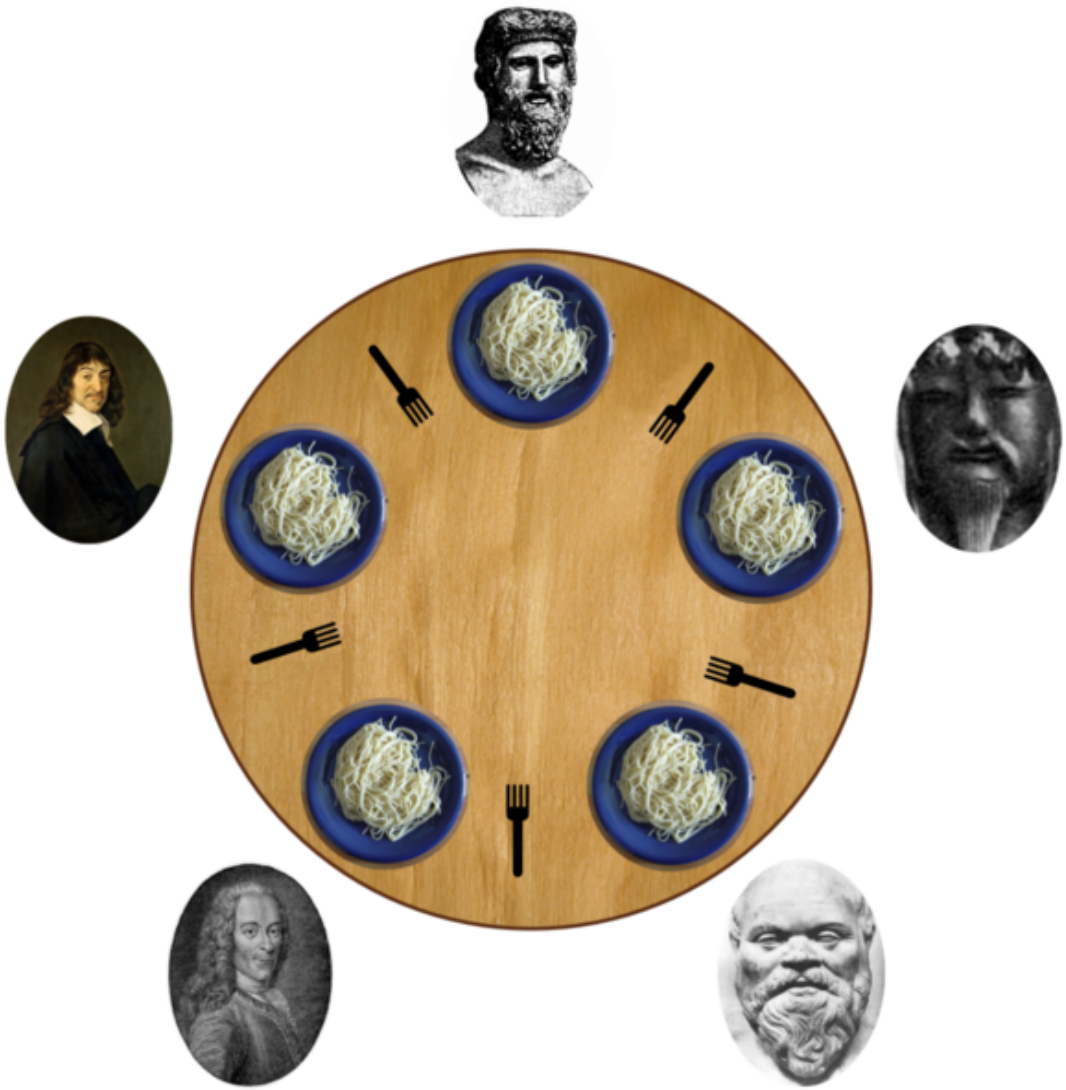

Bild: Benjamin D. Esham, Wikipedia

Implementierung

```
import java.util.concurrent.locks.Lock;

public class Philosoph extends Thread {
    private Lock links;
    private Lock rechts;

    public Philosoph(String name, Lock links, Lock rechts) {
        super(name);
        this.links = links;
        this.rechts = rechts;
    }

    private void debug(String message) {
        System.out.println(getName()+": "+message);
    }
```

Implementierung (Gabeln nehmen, hinlegen)

```
private void nehmeGabeln() throws InterruptedException {
    links.lock();
    // Zwischen dem Aufnehmen der Gabeln vergeht etwas Zeit
    Thread.sleep(5);
    rechts.lock();
}

private void legeGabelnHin() {
    links.unlock();
    rechts.unlock();
}
```

Implementierung (Essen, denken)

```
private void esse() throws InterruptedException {
    int dauer;
    // Esse für bis zu 100ms
    dauer = (int) (100*Math.random());
    debug("Essen für "+dauer+"ms");
    Thread.sleep(dauer);
}

private void denke() throws InterruptedException {
    int dauer;
    // Denke für bis zu 100ms
    dauer = (int) (100*Math.random());
    debug("Denken für "+dauer+"ms");
    Thread.sleep(dauer);
}
```

Implementierung (run)

```
    @Override
    public void run() {
        while (!isInterrupted()) {
            try {
                debug("Gabeln aufnehmen");
                nehmeGabeln();
                esse();
                debug("Gabeln hinlegen");
                legeGabelnHin();
                denke();
            } catch (InterruptedException e) {
                legeGabelnHin();
                interrupt();
            }
        }
    }
}
```

Analyse

Im einfachsten Fall kann man mit zwei Philosophen arbeiten, die sich gegenüber sitzen:

```
Lock l1 = new ReentrantLock();
Lock l2 = new ReentrantLock();

new Philosoph("P1",l1,l2).start();
Thread.sleep(30);
new Philosoph("P2",l2,l1).start();
```

Beim Ausführen bemerkt man, dass nach mehr oder weniger langer Zeit keiner der Philosophen mehr essen kann. Denn beide halten die linke Gabel in der Hand und warten auf die rechte – die dummerweise gerade vom anderen Philosophen festgehalten wird.

Lösungsmöglichkeit

Die Ursache für die Verklemmung in diesem Fall liegt in der Reihenfolge des Aufnehmens der Gabeln. Um diese aufzulösen, kann man eine *Goldene Gabel* einführen, die grundsätzlich zuerst aufgenommen werden muss, auch wenn sie rechts liegt.

Übrigens reicht auch bei mehr als zwei Philosophen eine einzige goldene Gabel aus.

Dies bedeutet:
Letztlich ist ein einziger Philosoph (der links von der goldenen Gabel) dafür verantwortlich, ob *alle* Philosophen verhungern müssen oder nicht!

Strategie zur Implementierung

Zur Lösung wird nur eine neue Klasse geschaffen:

```
public class GoldenLock extends ReentrantLock {
}
```

Diese dient rein zur Markierung einer Gabel als golden.

Nun wird in der Methode `nehmeGabeln()` sichergestellt, dass abweichend vom Normalfall die rechte Gabel zuerst aufgenommen wird, wenn sie aus Gold ist.

Änderung in nehmeGabeln()

```
private void nehmeGabeln() throws InterruptedException {
    Lock first;
    Lock second;
    if (rechts instanceof GoldenLock) {
        first = rechts;
        second = links;
    } else {
        first = links;
        second = rechts;
    }
    first.lock();
    // Zwischen dem Aufnehmen der Gabeln vergeht etwas Zeit
    Thread.sleep(20);
    second.lock();
}
```

Austausch von Signalen über Threads

In den bisherigen Beispielen wurde direkt mit den Lock-Objekten gearbeitet.

Zusätzlich existiert aber die Möglichkeit, dass Threads sich gezielt schlafen legen können, bis sie von einem anderen Thread benachrichtigt werden, dass sie ihre Arbeit fortsetzen können.

Dies wird seit Java 1.0 über die Methoden `wait` und `notify` angeboten. Seit Java 5 kann man aber Objekte vom Typ `Condition` verwenden, die weitere Möglichkeiten und mehr Komfort bieten.

Das Interface `Condition`

Objekte, die `Condition` implementieren, stellen folgende Methoden zur Verfügung

- `await()` Diese Methode legt den aktuellen Thread schlafen und sorgt dafür, dass er wieder aufwacht, wenn er über das Condition-Objekt benachrichtigt wird.
- `signal()` weckt *einen* Thread auf, der auf das Condition-Objekt wartet.
- `signalAll()` benachrichtigt alle schlafenden Threads.

Condition-Objekte werden von Lock-Objekten erzeugt:

```
Lock lock = new ReentrantLock();
Condition cond = lock.newCondition();
```

Von `await` existieren zeitgesteuerte Varianten analog zu `tryLock`.

Vorgehen

- Erzeugen eines Lock-Objekts `lock`
- Erzeugen eines Condition-Objekts:
  `Condition cond = lock.newCondition()`
- Threads dürfen nur dann in Warteposition geschoben werden, wenn sie den Lock besitzen. Es muss also zunächst `lock.lock()` aufgerufen werden!
- Mit `cond.await()` wird gewartet (und der Lock wieder freigegeben!), bis der Thread von einem anderen mit `cond.signal()` bzw. `cond.signalAll()` benachrichtigt wird.
- Der Thread wartet nun, bis er den Lock wieder hat und setzt die Arbeit fort.
- Am Ende das `lock.unlock()` nicht vergessen!

Beispiel: Erzeuger und Verbraucher

Im folgenden Beispiel soll ein System aus Erzeugern und Verbrauchern realisiert werden.

- Zentraler Punkt ist eine Warteschlange, die Daten aufnehmen und wieder abgeben kann.
- Mehrere Erzeuger können dort Daten abliefern.
- Mehrere Verbraucher können Daten abholen.
- Falls die Schlange voll ist, warten die Erzeuger, bis ein Verbraucher Nachricht gibt, dass er ein Element entfernt hat.
- Ist die Schlange leer, warten die Verbraucher, bis ein Erzeuger Nachricht gibt, dass er ein Element erzeugt hat.

Klasse WarteSchlange

```
public class WarteSchlange {
    private final int MAX = 10;
    private Queue<String> daten = new LinkedList<String>();

    Lock lock = new ReentrantLock();
    Condition nichtVoll = lock.newCondition();
    Condition nichtLeer = lock.newCondition();

    public void fuelle(String s) throws InterruptedException {
        lock.lock();
        try {
            while (daten.size()==MAX) nichtVoll.await();
            daten.add(s);
            nichtLeer.signalAll();
        } finally {
            lock.unlock();
        }
    }
```

Klasse WarteSchlange (Forts.)

```
    public String leere() throws InterruptedException {
        lock.lock();
        try {
            while (daten.size()==0) nichtLeer.await();
            String result = daten.poll();
            nichtVoll.signalAll();
            return result;
        } finally {
            lock.unlock();
        }
    }
}
```

Klasse Erzeuger

```
public class Erzeuger extends Thread {
    private WarteSchlange q;
    public Erzeuger(String name, WarteSchlange q) {
        super(name);
        this.q = q;
    }

    @Override
    public void run() {
        while (!isInterrupted()) {
            try {
                Date date = new Date();
                String msg = String.format("%tT %10.8f",date,Math.
                System.out.println(getName()+" schreibt "+msg);
                q.fuelle(msg);
                Thread.sleep((int)(Math.random()*100));
            } catch (InterruptedException e) {
                interrupt();
            }
        }
```

Klasse Verbraucher

```
public class Verbraucher extends Thread {
    private WarteSchlange daten;
    public Verbraucher(String name, WarteSchlange daten) {
        super(name);
        this.daten = daten;
    }

    @Override
    public void run() {
        while (!isInterrupted()) {
            try {
                System.out.println(getName()+" erhielt "+daten.lee
                Thread.sleep((int) (Math.random()*100));
            } catch (InterruptedException e) {
                interrupt();
            }
        }
    }
}
```

Klasse Main

```
public class Main {
    private final static WarteSchlange q = new WarteSchlange();

    public static void main(String[] args) {
        new Erzeuger("E1",q).start();
        new Erzeuger("E2",q).start();

        new Verbraucher("V1",q).start();
        new Verbraucher("V2",q).start();
        new Verbraucher("V3",q).start();
    }
}
```

Analyse

Die beiden interessanten Punkte der Anwendung liegen an den Stellen, an denen die Erzeuger bzw. Verbraucher schlafen gelegt werden:

```
while (daten.size()==MAX) nichtVoll.await();
while (daten.size()==0) nichtLeer.await();
```

Warum **while** und nicht **if**?

Analyse (Forts.)

Wird ein Element verbraucht, so werden mit `nichtVoll.signalAll()` alle wartenden Erzeuger benachrichtigt, dass wieder ein Platz frei ist. Nun bekommt einer der Erzeuger zufällig als erstes den Lock wieder und erzeugt ein neues Element. Kommt dann der nächste Erzeuger wieder daran, kann es sein, dass trotz Signals die Schlange also immer noch voll ist, wenn zwischenzeitlich kein anderer Verbraucher dran kam.
Daher muss geprüft werden, ob der Thread nicht direkt weiter warten muss!

Die gleiche Logik gilt bei den Verbrauchern. Hier muss ebenfalls geprüft werden, ob das gerade angekommene Element nicht schon von einem anderen Verbraucher genommen wurde.

Reflections: Motivation

Im weiteren Verlauf des Semesters werden wir beim objekt-relationalen Mapping zwischen Java und Datenbanken mit Hibernate ein Mapping zwischen Tabellen-Spalten und Java-Klassen kennen lernen, das per XML-Datei definiert wird.

Hier werden zur Laufzeit anhand des Mappings

- Instanzen einer Klasse angelegt,
- Methoden (Getter und Setter) dieser Klasse aufgerufen,

obwohl erst nach Lesen der XML-Datei bekannt ist, wie die Klassen und Methoden heißen.

Wie ist es möglich, zur Laufzeit Klassen zu laden und diese inklusive deren Methoden zu benutzen, wenn deren Name überhaupt erst zur Laufzeit feststeht?

Reflections

Java bietet eine Eigenschaft, die ältere Programmiersprachen wie C++ meist nicht kennen

Reflections (dt.: Reflexionen)

Dies bedeutet: Java-Programme kennen ihre eigene Struktur, besitzen also zur Laufzeit Informationen über Klassen, deren Attribute und Methoden, sowie deren Sichtbarkeit, Typen, Rückgabewerte, Parameter etc.

Weitere Programmiersprachen, die Reflections unterstützen: C#, Visual Basic .NET, Lisp, Python, Ruby

Beispiel

Betrachten wir als Beispiel die Klasse `Person` aus dem letzten Praktikum. Wie kann man mit dieser Klasse in einer Anwendung so arbeiten, dass sie dem Compiler beim Kompilieren der Anwendung nicht bekannt sein muss?

Um dies zu können, dürfen die Klasse und die Methoden nicht direkt im Quelltext benutzt werden, sondern

- die Klasse muss zur Laufzeit anhand ihres Namens geladen werden,
- benötigte Methoden müssen ebenfalls anhand des Namens geladen und dann aufgerufen werden.

Beispiel-Code

```
import java.lang.reflect.Method;

public class Main {
    public static void main(String[] args) throws Exception {
        Class<?> personClass = Class.forName("person.Person");
        Object person = personClass.newInstance();
        Method getNachname = personClass.getMethod("getNachname");
        Class<?> nachnameClass = getNachname.getReturnType();
        Method setNachname =
            personClass.getMethod("setNachname", nachnameClass);
        System.out.println(nachnameClass.getName());
        setNachname.invoke(person, nachnameClass.cast("Meier"));
        System.out.println(getNachname.invoke(person));
    }
}
```

Analyse

- Die Klasse `Person` wird anhand des Names geladen.
- Von dieser wird mit `newInstance` ein Objekt mit dem Standardkonstruktor instanziert.
- Anschließend wird eine Methode ohne Parameter anhand des Namens `getNachname` geladen.
- Der Rückgabetyp dieser Methode wird mit `getReturnType()` abgefragt.
- Nun wird die Methode mit dem Namen `setNachname` geladen, die einen Parameter dieses Typs als Parameter verlangt.
- Nun wird zunächst der so geladene Setter mit einem Namen aufgerufen und anschließend dieser Nachname mit dem Getter zurückgelesen und ausgegeben.

Die Klasse `Class`

`Class` besitzt als statische Methode `forName`, die anhand eines Klassennamens ein Klassenobjekt zurück gibt.

Übrigens kann man für jedes Objekt die Methode `getClass()` verwenden, um festzustellen, von welcher Klasse es instanziert wurde, z. B.

```
if (o.getClass().equals(String.class)) { ... }
```

Den Namen eines Klassenobjekts erhält man mit der Methode `getName()`, weitere Methoden sind z. B. `isArray()` (handelt es sich um einen Array?), `isInterface()` (handelt es sich um ein Interface?), `isPrimitive()` (handelt es sich um einen primitiven Typ wie `int`?).

Methoden

Die Methode `getMethods()` von `Class` liefert einen Array `Method[]` aller öffentlichen Methoden, die Methode `getMethod(String name, Class c1, ...)` eine Instanz von `Method` mit der Parameter-Signatur `c1, ...`.

`c.getMethod("foo")` sucht also die Methode mit leerer Parameterliste,
`c.getMethod("foo",String.class,int.class)` die Methode mit einem `String` und einem `int` als Parameter.

Informationen über die Methode erhält man mit

`String getName()` Name der Methode,

`Class<?> getReturnType()` Rückgabetyp,

`Class<?>[] getParameterTypes()` Array der Parametertypen.

JavaBeans

JavaBeans sind normale Java-Klassen, die aber eine standardisierte Form besitzen:

- Es existiert ein Standardkonstruktor,
- Es existieren für die Attribute Getter und Setter, diese heißen `void setXXX(T val)` und `T getXXX()` (bei boolean: `boolean isXXX()`).

Diese standardisierte Form ermöglicht das Anlegen mit `Class.newInstance()` und erleichtert das Suchen von Methoden über die Reflections.

Solche Klassen können daher einfach in anderen Frameworks verwendet werden, z. B. GUI-Bilder, objekt-relationales Mapping mit JPA, XML-Binding mit JAXB.

Beispiel

```
import java.beans.*;
import java.lang.reflect.Method;
import java.util.*;

public class Main {
    public static void main(String[] args) throws Exception {
        Class<?> personClass = Class.forName("person.Person");
        BeanInfo bi = Introspector.getBeanInfo(personClass);
        Map<String,PropertyDescriptor> props =
            new HashMap<String,PropertyDescriptor>();

        for (PropertyDescriptor desc : bi.getPropertyDescriptors()) {
            props.put(desc.getName(), desc);
        }

        Object person = personClass.newInstance();
        Method setter = props.get("nachname").getWriteMethod();
        Method getter = props.get("nachname").getReadMethod();
        setter.invoke(person, setter.getParameterTypes()[0].cast("Meier"))
        System.out.println(getter.invoke(person));
    }
}
```