Nebenläufigkeit: wait() und notify()

- „Producer/Consumer“-Problem verstehen und lösen können
- Monitore mit Bedingungssynchronisation anwenden können
- Unterschied zwischen notify() und notifyAll() verstehen und anwenden können
- Unterschied zwischen wait() und sleep() verstehen
- Begriff Deadlock einordnen können
- „Dining Philosophers“-Problem verstehen und programmtechnisch umsetzen können
- Deadlock-Vermeidungsstrategien implementieren können

Einführendes Beispiel „Producer/Consumer“-Problem

Producer

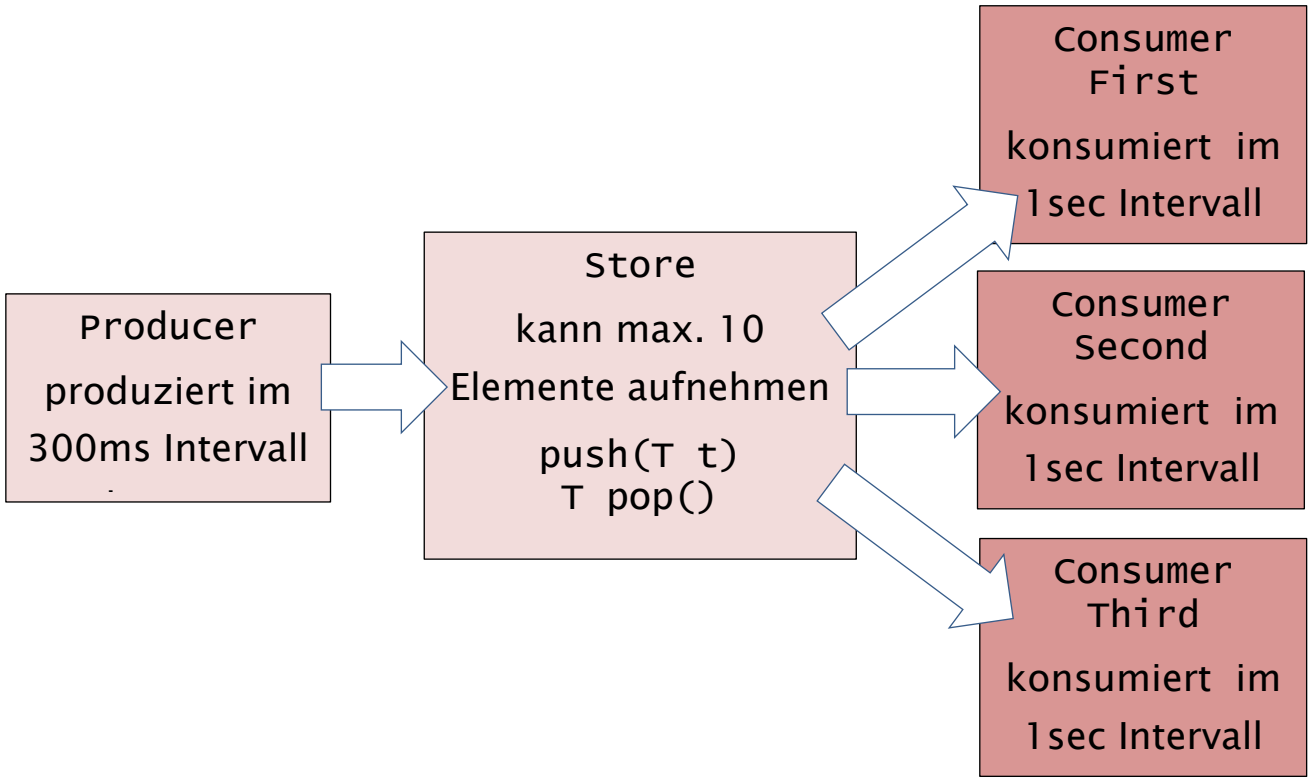
- Liefert Daten und teilt das Vorhandensein dem Consumer mit (notify())
- Wartet bis Consumer Daten verarbeitet hat (wait())

Consumer

- Wartet (wait()) auf Daten und verarbeitet diese
- Teilt Producer mit dass er diese verarbeitet hat (notify())

WICHTIG

- Gleichzeitiger Zugriff auf Daten muss unterbunden werden (**synchronized**)



```
public class Store<T>
{
  public static final int MAX_QUEUE = 10;

  private Queue<T> items = new LinkedList<T>();

  public synchronized void push(T item) {
    while (items.size() == MAX_QUEUE)
      try {
        wait();
      } catch (InterruptedException e) { ; }
    items.add(item);
    notify();
  }

  public synchronized T pop() {
    T ret = null;
    while (items.size() == 0)
      try {
        wait();
      } catch (InterruptedException e) { ; }
    ret = items.poll();
    notify();
    return ret;
  }

  public synchronized int getSize() {
    return items.size();
  }
}
```

```
public class Producer extends Thread
{
  public static final int MAX_SLEEP = 300;

  private Store<String> store = null;

  public Producer(Store<String> store) {
    this.store = store;
    start();
  }

  @Override
  public void run() {
    SimpleDateFormat sdf =
      new SimpleDateFormat("HH:mm:ss SSS");
    while (true) {
      String s = sdf.format(new Date());
      store.push(s);
      System.out.println("Size "+store.getSize()+
        " produced "+s);
      try {
        Thread.sleep(
          (int)(Math.random() * MAX_SLEEP));
      } catch (InterruptedException e) { ; }
    }
  }
}
```

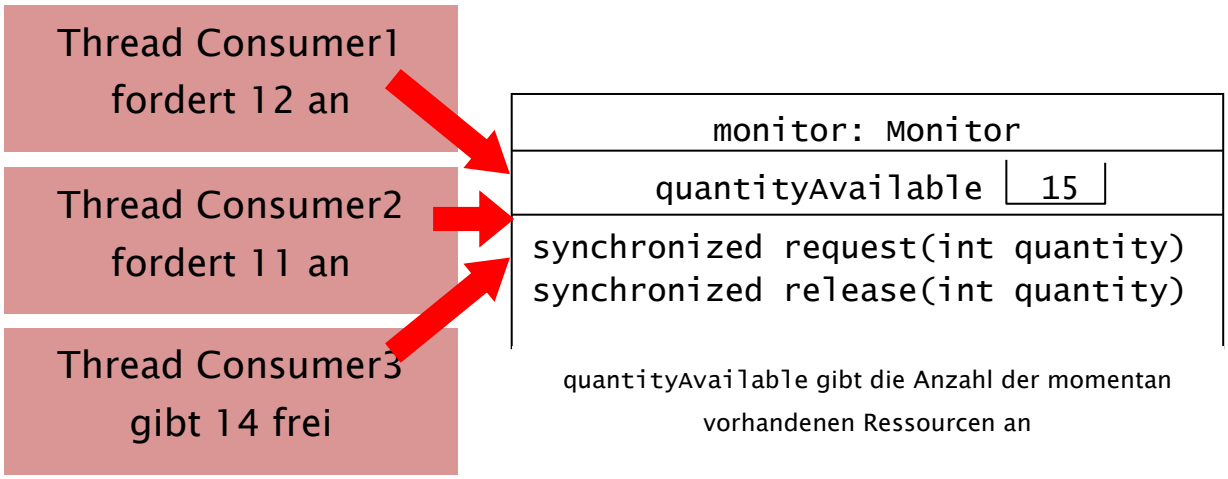
```
public class Consumer extends Thread
{
  public static final int MAX_SLEEP = 1000;

  private Store<String> store = null;

  public Consumer(Store<String> store, String name){
    this.store = store;
    this.setName(name);
    start();
  }

  @Override
  public void run() {
    while (true) {
      String s = store.pop();
      System.out.println(getName()+" consumes "+s);
      try {
        Thread.sleep(
          (int)(Math.random() * MAX_SLEEP));
      } catch (InterruptedException e) { ; }
    }
  }
}
```

```
Store<String> store = new Store<String>();

new Producer(store);

new Consumer(store, "First");
new Consumer(store, "Second");
new Consumer(store, "Third");
```

```
Size 1 produced 09:51:31 763
Third consumes 09:51:31 763
Size 1 produced 09:51:31 848
Second consumes 09:51:31 848
Size 1 produced 09:51:31 908
Size 2 produced 09:51:31 996
Size 3 produced 09:51:32 061
Size 4 produced 09:51:32 138
Size 5 produced 09:51:32 156
Size 6 produced 09:51:32 230
First consumes 09:51:31 908
Size 6 produced 09:51:32 299
Size 7 produced 09:51:32 377
Size 8 produced 09:51:32 468
Size 9 produced 09:51:32 502
Size 10 produced 09:51:32 578
Second consumes 09:51:31 996
Size 10 produced 09:51:32 606
Third consumes 09:51:32 061
Size 10 produced 09:51:32 636
Second consumes 09:51:32 138
Size 10 produced 09:51:32 795
```

Monitore mit Bedingungssynchronisation

Ein Thread (Consumer) wartet bis eine Bedingung erfüllt ist – verursacht durch einen anderen Thread (anderer Consumer)



...

wait() gibt zum Unterschied zu sleep() Monitor frei

Anwendungsbeispiele

- Erst dann in Variable schreiben, wenn diese bestimmten Wert hat
- Walkman-Vergabe an Benutzergruppen im Museum
- Parkplatzvergabe

Consumer der in einem kritischen Abschnitt auf Bedingung wartet, muss Monitor frei geben (wait()), damit andere Threads kritischen Abschnitt durcharbeiten und Zustand ändern können

Thread der Zustand ändert, informiert mit

- notify()
  nur einen wartenden Thread der zufällig ausgewählt wird, oder mit
- notifyAll()
  alle wartenden Threads

wait(), notify() und notifyAll() können nur in synchronisierten Methoden oder Blöcken aufgerufen werden

ACHTUNG!!!

Würde im obigen Beispiel anstelle von notifyAll() nur notify() aufgerufen, besteht die Gefahr, dass nach dem Freigeben falscher Consumer informiert wird, der keine Reservierung vornimmt, weil seine geforderte Anzahl zu hoch ist ➔ notify() „verpufft“

Konkret

```
public class Monitor
{
  public static final int MAX_AVAILABLE = 20;

  private int quantityAvailable = MAX_AVAILABLE;

  public synchronized void request(int quantity) {
    while (quantityAvailable < quantity)
      try {
        wait();
      } catch (InterruptedException e) { ; }
    quantityAvailable-=quantity;
    System.out.println(
      Thread.currentThread().getName() +
      " receives " + quantity +
      " ressources. Now available " + quantityAvailable);
    notityAll();
  }

  public synchronized void release(int quantity) {
    quantityAvailable+=quantity;
    System.out.println(
      Thread.currentThread().getName() +
      " releases " + quantity +
      " ressources. Now available " + quantityAvailable);
    notifyAll();
  }
}
```

```
public class ConsumerMain
{
  public static void main(String[] args) {
    final Monitor m = new Monitor();

    for (int i = 0; i < 5; i++) {
      new Thread() {
        @Override
        public void run() {
          for (int i = 0; i < 4; i++) {
            int quantity = (int)(Math.random() *
              Monitor.MAX_AVAILABLE) + 1;
            m.request(quantity);
            try {
              sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) { ; }
            m.release(quantity);
          }
        }
      }.start();
    }
  }
}
```

```
Thread-0 receives 13 ressources. Now available 7
Thread-3 receives 6 ressources. Now available 1
Thread-0 releases 13 ressources. Now available 14
Thread-4 receives 7 ressources. Now available 7
Thread-1 receives 2 ressources. Now available 5
Thread-3 releases 6 ressources. Now available 11
Thread-4 releases 7 ressources. Now available 18
Thread-3 receives 17 ressources. Now available 1
Thread-1 releases 2 ressources. Now available 3
Thread-1 receives 3 ressources. Now available 0
Thread-1 releases 3 ressources. Now available 3
Thread-1 receives 2 ressources. Now available 1
Thread-3 releases 17 ressources. Now available 18
Thread-4 receives 7 ressources. Now available 11
Thread-4 releases 7 ressources. Now available 18
Thread-3 receives 18 ressources. Now available 0
Thread-1 releases 2 ressources. Now available 2
Thread-3 releases 18 ressources. Now available 20
Thread-1 receives 5 ressources. Now available 15
```

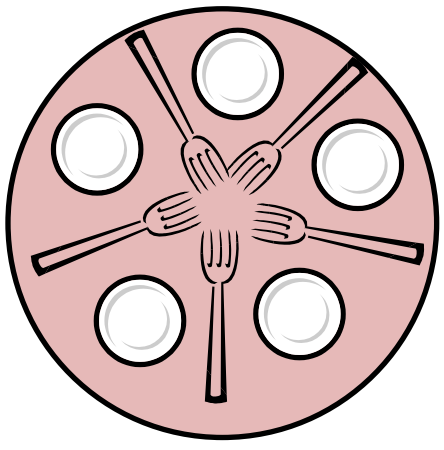
Deadlock (dtsch. Verklemmung)

Thread A belegt Ressource die Thread B haben möchte. Thread B belegt Ressource die Thread A haben möchte. Beide Threads brauchen beide Ressourcen um weiterzumachen. Sie warten…

Beispiel „Dining Philosophers" Problem[1]

Fünf Philosophen sitzen an einem runden Tisch und jeder hat einen Teller mit Essen vor sich. Zum Essen benötigt jeder Philosoph zwei Gabeln. Allerdings sind am Tisch nur fünf Gabeln vorhanden, die zwischen den Tellern liegen. Die Philosophen können also nicht gleichzeitig speisen.

Die Philosophen denken über philosophische Probleme nach. Wenn einer hungrig wird, greift er zuerst die Gabel rechts von seinem Teller, dann die auf der linken Seite und beginnt zu essen. Wenn er satt ist, legt er die Gabeln wieder zurück und beginnt wieder zu denken. Sollte eine Gabel nicht an ihrem Platz liegen, wenn der Philosoph sie aufnehmen möchte, so wartet er, bis die Gabel wieder verfügbar ist.



Programmstruktur

```
while (true) {
   denken;
   nimm rechte Gabel;
   nimm linke Gabel;
   essen
   gib rechte Gabel;
   gib linke Gabel;
}
```

Verklemmung

Alle Philosophen haben rechte Gabel und warten auf die linke Gabel

[1] von E.W. Dijkstra, Wegbereiter der Strukturierten Programmierung

Konkret

```
public class Philosopher extends Thread
{
  public static final int MAX_THINK_TIME = 2000;
  public static final int MAX_EAT_TIME = 1000;

  private Fork left, right = null;

  public Philosopher(String name, Fork left, Fork right) {
    setName(name); this.left = left; this.right = right;
  }

  @Override
  public void run() {
    while (true) {
      try {
        sleep((int)(Math.random() * MAX_THINK_TIME));
      } catch (InterruptedException e) { ;      }
      right.get(this);
      left.get(this);
      try {
        sleep((int)(Math.random() * MAX_EAT_TIME));
      } catch (InterruptedException e) { ; }
      right.put(this);
      left.put(this);
    }
  }
}
```

```
Fork f1 = new Fork("F1");
Fork f2 = new Fork("F2");
Fork f3 = new Fork("F3");

Philosopher p1 = new Philosopher("P1", f3, f1);
Philosopher p2 = new Philosopher("P2", f1, f2);
Philosopher p3 = new Philosopher("P3", f2, f3);

p1.start(); p2.start(); p3.start();
```

```
public class Fork
{
  private String name = null;;
  private boolean available = true;

  public Fork(String name) {
    this.name = name;
  }

  public synchronized void get(Philosopher p){
    while (!available)
      try {
        wait();
      } catch (InterruptedException e) { ; }
    available = false;
    System.out.println(p.getName() +
      " gets " + name);
    notifyAll();
  }

  public synchronized void put(Philosopher p){
    available = true;
    System.out.println(p.getName() +
      " puts " + name);
    notifyAll();
  }
}
```

```
P2 gets F1
P1 puts F3
P3 gets F3
P2 puts F2
P3 gets F2
P2 puts F1
P1 gets F1
P3 puts F3
P1 gets F3
P3 puts F2
P2 gets F2
P1 puts F1
P2 gets F1
P1 puts F3
P2 puts F2
P2 puts F1
P3 gets F3
P2 gets F2
P1 gets F1
```

1. Lösungsmöglichkeit

Einer der Threads z.B. P1 nimmt zuerst die linke und dann erst die rechte Gabel

2. Lösungsmöglichkeit

- Ressourcen zugleich anfordern – nicht nacheinander
- Thread erhält entweder beide Ressourcen oder keine
- Wenn Thread eine Ressource hat und erkennt dass andere bereits belegt ist, gibt er seine bereits belegte Ressource frei

Konkret

- Klasse ForkControl enthält get() und put(). Diese Methoden greifen beide Gabel in einer Operation wenn diese verfügbar sind, sonst wird gewartet
- Pilosophen verwenden nur mehr Methoden von ForkControl und nicht mehr jene von Fork