Informatik B - Objektorientierte Programmierung in Java

Vorlesung 21:
Threads 2

Inhalt

- Threads 2
  - Synchronisation mit **`wait`** und **`notify`**
  - Producer/Consumer-Beispiele
  - Lebenszyklus eines Threads
  - Probleme mit Threads
    - Deadlocks
    - Race Condition
  - Ergänzungen
    - **`synchronized`**
    - Thread-Namen
    - **`yield()`**
    - Priorität
    - Dämonen

`wait` und `notify`

- Neben dem Monitorkonzept stehen mit den Methoden **`wait`** und **`notify`** der Klasse **`Object`** noch weitere Synchronisationsprimitive zur Verfügung. Zusätzlich zu der bereits erwähnten Sperre, die einem Objekt zugeordnet ist, besitzt das Objekt nämlich auch noch eine *Warteliste*. Dabei handelt es sich um eine (möglicherweise leere) Menge von Threads, die auf ein Ereignis warten, um fortgesetzt werden zu können.

`wait` und `notify`

- Sowohl **`wait`** als auch **`notify`** dürfen nur aufgerufen werden, wenn das Objekt bereits gesperrt ist, also nur innerhalb eines **`synchronized`**-Blocks für dieses Objekt.
- Ein Aufruf von **`wait`** nimmt die bereits gewährte Sperre (temporär) zurück und stellt den Thread, der den Aufruf von **`wait`** verursachte, in die Warteliste des Objekts. Dadurch wird er unterbrochen und im Scheduler als *wartend* markiert.
- Ein Aufruf von **`notify`** weckt einen (beliebigen) Thread aus der Warteliste des Objekts und führt ihn dem normalen Scheduling zu. Danach versucht der Thread, den Monitor zu belegen, um seine unterbrochene Aufgabe im **`synchronized`**-Block fortzusetzen.

Einsatz von `wait` und `notify`: Producer/Consumer

- Beispiel: Listing2213.java
- Ein Prozess arbeitet dabei als Produzent, der Fließkommazahlen »herstellt«, und ein anderer als Konsument, der die produzierten Daten »verbraucht«. Die Kommunikation zwischen beiden erfolgt über ein gemeinsam verwendetes **`Vector`**-Objekt, das die produzierten Elemente zwischenspeichert und als Monitor für die **`wait`**-/**`notify`**-Aufrufe dient.
- Da die Wartezeit zufällig ausgewählt wird, kann es durchaus dazu kommen, dass der Produzent eine größere Anzahl an Elementen anhäuft, die der Konsument noch nicht abgeholt hat.

Producer/Consumer

- Ein Thread arbeitet als Produzent, der Fließkommazahlen »herstellt«, und ein anderer als Konsument, der die produzierten Daten »verbraucht«. Die Kommunikation zwischen beiden erfolgt über ein gemeinsam verwendetes **`Vector`**-Objekt, das die produzierten Elemente zwischenspeichert: das Lager.
- Diese Referenz auf das Lager wird als Monitor zum Sperren des exklusiven Zugriffs auf das Lager verwendet.

Einsatz von `wait` und `notify`: Producer/Consumer

- Da der Konsumenten-Thread nur dann sinnvoll arbeiten kann, wenn auch Daten im Lager vorhanden sind, reicht es nicht aus, das gleichzeitige Ablaufen von Entfernen und Hinzufügen über **`synchronized`** zu verhindern. Der Konsumenten-Thread würde sonst regelmäßig versuchen, aus dem leeren Lager Daten zu entnehmen. Man benötigt also eine Möglichkeit, im Falle eines leeren Puffers den Konsumenten-Thread auf das Eintreffen neuer Daten warten zu lassen. Dies erreicht man mit der Methode **`wait`**.

Beispiel: P/C (1)

- ProducerComsumer.java

Producer

```
while (true) {
     s = "Wert "+Math.random();
     synchronized (v) {
       v.addElement(s);
       System.out.println("Produzent erzeugte "+s);
       v.notify();
     }...
```

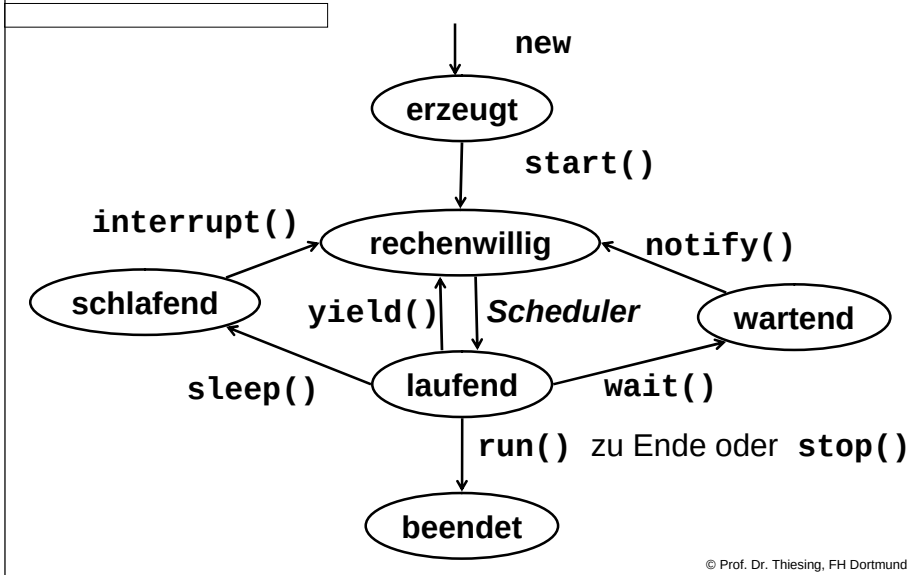
Consumer

```
while (true) {
        synchronized (v) {
           while (v.size() < 1) {
              try {
                 v.wait();
              } catch (InterruptedException e) {
                 //nichts
              }
           }
           v.removeElementAt(0);
        }...
```

Funktionsweise von `notify`

- **`notify`** weckt einen **beliebigen** Thread aus der Warteliste des Objekts, aber nicht einen bestimmten.
- Der aufgeweckte Thread kann solange nicht im synchronisierten Block fortfahren bis der darin befindliche Thread den Monitor abgibt. Danach versucht er (gleichberechtigt mit anderen Threads) den Monitor zu ergattern. Insbesondere genießt der gerade aufgeweckte Thread keinerlei Vorzugsbehandlung.

Lebenszyklus eines Threads



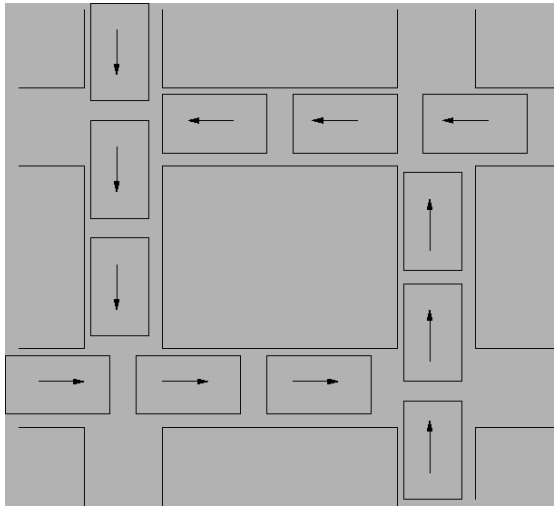
`wait()` vs. `sleep()`

- Es ist wichtig, dass der Konsument mit **`wait()`** darauf wartet, dass der Puffer befüllt wird, denn dadurch gibt er die Sperre frei.
- Würde der Konsument statt dessen **`Thread.sleep()`** ausführen, würde er mit dem Monitior in der Hand schlafen und die Sperre nicht frei geben: Der Produzent könnte das Lager nicht befüllen und bis in alle Ewigkeit darauf warten.
- Beispiel: ProducerConsumerMitSleep.java

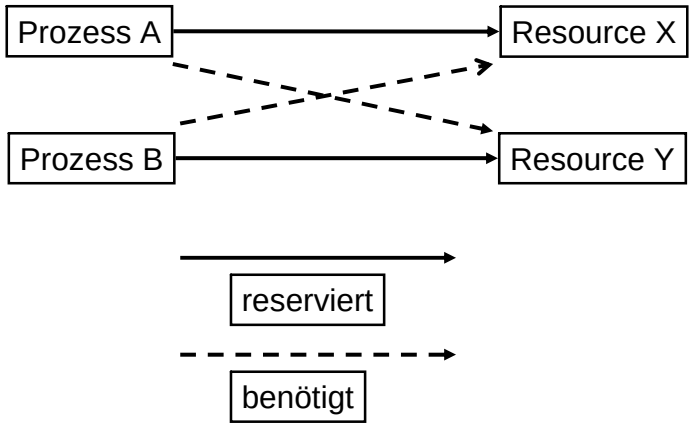
Definition: Deadlock

- Eine Situation in der Prozessausführung, in der zwei oder mehr Prozesse nicht weiterarbeiten können, weil jeder darauf wartet, dass mindestens ein anderer etwas bestimmtes erledigt.
- Deadlock = Verklemmung des Prozesses

Beispiel: Deadlock im Verkehr

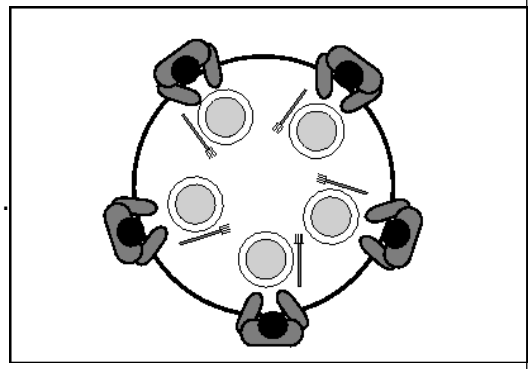


Entstehung von Deadlocks i.A.



Standardbeispiel: Dining Philosophers

- 5 Philosophen brauchen zum Essen je zwei Gabeln, aber es existieren nur 5.
- Wenn alle 5 zur Gabel zu ihrer Rechten greifen, entsteht ein Deadlock.
- Zweites Problem: Aushungern eines Philosophen. Kann mit kritischen Abschnitten nicht verhindert werden. Java erlaubt einem *beliebigen* Thread, dranzukommen.



Deadlocks können auftreten, wenn jede der folgenden Bedingungen eintritt:

1. Betriebsmittel werden exklusiv von genau einem Prozess belegt oder sind frei (Gabel).
2. Prozesse warten auf die Zuteilung weiterer Betriebsmittel, während sie bereits welche belegen (linke nach rechter Gabel).
3. Betriebsmittel können nicht entzogen werden, sondern werden vom Prozess zurückgegeben (Hinlegen einer Gabel).
4. Es besteht ein Ring aus zwei oder mehr Prozessen, bei dem jeder auf ein von einem anderen belegtes Betriebsmittel wartet (5 Philosophen).

Strategien zur Deadlock-Bekämpfung

- Erkennen und Auflösen zur Laufzeit
- Vermeidung:
  - Eine der 4 Bedingungen durch Einschränkung ausschalten.
- Verhinderung
  - Eine der 4 Bedingungen durch Zusatzinformation ausschalten.
  - 2. *Halten und Warten*
    - (a) Reserviere alles im voraus.
    - (b) Gib alles vor der nächsten Reservierung zurück.
  - 4. *Zyklisches Warten*
    - Totale Ordnung auf alle Ressourcen
    - Reservierung nur in bestimmter Reihenfolge (z.B. aufsteigend)

Beispiel: P/C (2)

- ProducerConsumerLimited.java
- Ein Produzent und ein Konsument mit einem begrenzten Lager.

Producer

```
synchronized (v) {
  while (v.size() == limit) {
     try {
       v.wait();
     } catch (InterruptedException e) {
       //nichts
     }
  }
  v.addElement(s);
  System.out.println("      Produzent erzeugte "+s);
  v.notify();
 }
```


Beispiel: Erläuterung

- ProducerConsumerLimited.java
- Wenn das Lager voll ist, wird der Produzent wartend. Aus diesem Zustand wird er nie erlöst, denn der Konsument macht kein **`notify()`**, ist aber selber wartend nachdem er das Lager geleert hat. Der Prozess hängt bis in alle Ewigkeit.

Beispiel: Lösung

- ProducerConsumerLimitedWithoutDeadlock.java

Consumer

```
while (true) {
  synchronized (v) {
     while (v.size() < 1) {
        try {
           v.wait();
        } catch (InterruptedException e) {
           //nichts
        }
     }
     System.out.print(
       "Konsument fand "+(String)v.elementAt(0)
     );
     v.removeElementAt(0);
     System.out.println(" (verbleiben:+v.size()+")");
     v.notify();
  }
```

Beispiel: P/C (3)

- ManyProducerManyConsumerLimited.java
- Mehrere Produzenten und mehrere Konsumenten mit begrenztem Lager.
- Die **`run`**-Methoden bleiben im Wesentlichen unverändert.
- Ausschnitt ais **`main():`**

```
Consumer[] c = new Consumer[ANZAHL_CONSUMER];
  for (int i = 0; i < ANZAHL_CONSUMER; i++)
      {
          c[i] = new Consumer(v, i);
          c[i].start();
      }...
```


Beispiel: Erläuterung

- ManyProducerManyConsumerLimited.java
- Die Synchronisation mit **`wait()`** und **`notify()`** funktioniert für mehrere Produzenten- und Konsumenten-Threads nicht zuverlässig, sondern es kann ein Deadlock eintreten
- Beispiel mit 2 Produzenten, 2 Konsumenten, Lagergröße 1:

```
Konsument 1 is waiting
                    Produzent 0 erzeugte Wert 0.05507594124773707
                    Produzent 1 is waiting
                    Produzent 0 is waiting
Konsument 0 fand Wert 0.05507594124773707 (verbleiben: 0)
Konsument 1 is waiting
Konsument 0 is waiting
```


Beispiel: Erläuterung

- ManyProducerManyConsumerLimited.java
- Begründung für Deadlock:
  In der Warteliste befinden sich Konsumenten- und Produzenten-Threads. Durch **`notify()`** wird ein **beliebiger** Thread aus der Warteliste gelöst.
- Im Beispiel warten bereits die beiden Produzenten und Konsument 1. Konsument 0 leert das Lager, durch sein **`notify()`** wird zufällig Konsument 1 aus der Warteliste befreit, aber wegen des leeren Lagers müssen anschließend beide Konsumenten-Threads warten: Alle 4 Threads hängen in der Warteliste.

Lösung: `notifyAll`

- **`notify`** holt einen **beliebigen** Thread aus der Warteliste des Objekts, aber nicht einen bestimmten
- Aber **`notifyAll`** holt **alle** wartenden Threads aus der Warteliste!

Beispiel: P/C (4)

- ManyProducerManyConsumerLimitedWithoutDeadlock.java
- **`run`**-Methoden werden verändert:
  `v.notifyAll();`
  statt
  `v.notify();`
- Dadurch werden jetzt **alle** wartenden Threads aus der Warteliste geholt, insbesondere auch solche von der anderen Klasse.
- D.h. ein Konsument holt sicher einen Produzenten aus der Warteliste, der gebraucht wird, um das Lager wieder zu befüllen, damit ein Konsument weiterarbeiten kann und umgekehrt.

Thread-Probleme: Race-Condition

- Eine sog. *Race-Condition* ergibt sich, wenn das Ergebnis einer Berechnung von der Ausführungsreihenfolge der Prozesse abhängt.
- Problem: zwei (oder mehr) Threads sollen in bestimmter Reihenfolge auf eine Ressource zugreifen.
- Dies muss synchronisiert werden und darf nicht vom Zufall des Schedulers abhängen.

Ergänzung: `synchronized`-Methoden

- Problem: Es müssen zwei Methoden eines Objekts exklusiv bearbeitet werden, nicht nur eine!
- Lösung: Der Monitor für eine Methode hängt an dem Objekt, auf das die Methode wirkt
  => auch bei mehreren Methoden, die auf dasselbe Objekt wirken, erfolgt damit ein gegenseitiger Ausschluss.
- Besitzt ein Thread schon den Monitor eines Objekts - d.h. hat er bereits eine **`synchronized`**-Methode betreten -, so kann *er* dennoch *weitere* **`synchronized`**-Methoden *desselben* Objekts aufrufen.

Thread-Namen

- Thread-Name setzen:

```
Thread th1 = new Thread();
th1.setName("Thread 1>>> ");
```

- Thread-Name in **`run()`** abfragen:

```
System.out.print(
  Thread.currentThread().getName());
```

- Siehe folgendes Beispiel

yield()

- „yield“ = „abgeben“
- Durch Aufruf von **`yield()`** kann ein Thread selbst und freiwillig den Prozessor abgeben, so dass der Scheduler möglicherweise einem anderen Thread Rechenzeit zuteilt.
- Beispiel: ThreadYield.java

```
Thread.yield();
```

- Führt dazu, dass die gleichpriorisierten Threads überwiegend abwechselnd drankommen.

Priorität

- Die Priorität steuert den Scheduler in der Weise, dass bei Vorhandensein mehrerer bereiter Threads diejenigen mit höherer Priorität vor denen mit niedrigerer Priorität ausgeführt werden.
- Als Parameter kann an **`setPriority()`** ein Wert übergeben werden, der zwischen den Konstanten **`Thread.MIN_PRIORITY`** und **`Thread.MAX_PRIORITY`** liegt. Der Normalwert wird durch die Konstante **`Thread.NORM_PRIORITY`** festgelegt. In der aktuellen Version des JDK haben diese Konstanten die Werte 1, 5 und 10.

Priorität

- Beispiel: ThreadPriority.java
- Durch setzen der Prioritäten der beiden Threads auf 6 und 4 wird trotz **`Thread.yield()`** zunächst nur der höherpriorisierte ausgeführt.

Dämon

- ein im Hintergrund laufender Thread
- wartet in der Regel auf Aufträge von außen
- Beispiele: Drucker-Spooling
- Methoden:
  - **`public void setDaemon( boolean )`** macht einen Thread zum Dämon
  - **`public boolean isDaemon()`** gibt an, ob ein Thread ein Dämon ist

Dämonen

- Um einen Dämon zu erzeugen, muss nach dem Erzeugen, aber vor dem Starten des Threads der Aufruf **`setDaemon(true)`** erfolgen.
- Jedes Java-Programm hat Dämon-Threads, die automatisch von der Laufzeitumgebung gestartet werden. Der wichtigste von ihnen ist der Garbage Collector.

Ende eines Java-Prozesses

- Ein Java-Prozess endet, wenn der letzte **nicht-dämonische** Thread des Java-Prozesses beendet wird.
- Durch Aufruf von **`System.exit()`** lassen sich Prozesse mit laufenden Vordergrund-Threads abbrechen.