Threads

in

Java

ein

kleines

Skript

Threads in Java und anderswo 3

Ein konkretes Beispiel (Pi-berechnung, Gui-Reaktion): 4

Thread-Zustände und Zeitscheibenverteilung. 6

Thread-Api 7

Timer 8

Thread synchronization und Timer-Implementation 10

- **synchronized:** 11
- **wait und notify** 11
- **Timer-Implementierung:** 12

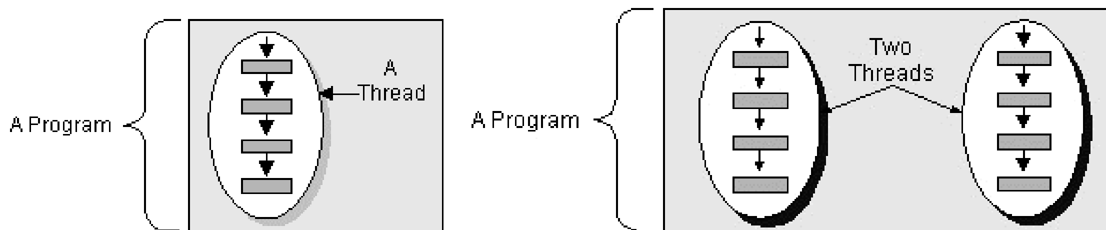
Deadlocks/Verklemmungen 15

Programme 20

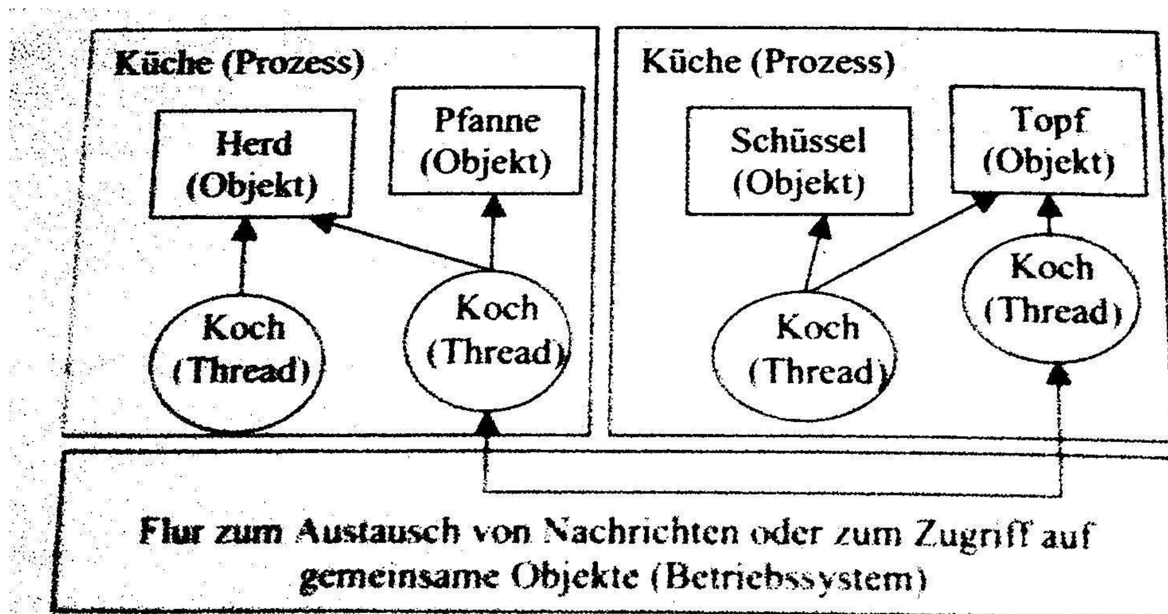
Literatur 20

1. Threads in Java und anderswo

Threads sind eine Möglichkeit Programmabläufe zu parallelisieren. Bisher sind Programme für ja ausschließlich sequentiell (bzw. wenn Ereignis gesteuert abschnittsweise sequentiell) abgelaufen. Threads ermöglichen es nun mehrere sequentielle Abläufe gleichzeitig ablaufen zu lassen. Der Name Thread kommt aus dem Englischen und bedeutet Bindfaden („roter Faden“), man kann sich einen sequentiellen Programmablauf bildlich als eine Reihe von Anweisungen vorstellen, die auf einem Bindfaden aufgefädelt sind. Ein paralleler Ablauf wird dann durch mehrere Bindfäden nebeneinander symbolisiert, diese arbeiteten einzeln wieder sequentiell, aber eben gleichzeitig und unabhängig von einander.



In gewisser Weise entsprechen Threads in einem Programm den Prozessen in einem Multitaskingbetriebssystem wie Unix oder Windows NT/2000/XP. Ein wichtiger Unterschied ist jedoch das sich mehrere Threads innerhalb eines Programms den globalen Speicherbereich (Heap) teilen, dies ist für Prozesse, die über komplett getrennte Adressräume im Speicher verfügen, nicht der Fall. Wegen ihrer ansonsten jedoch ähnlichen Funktion werden Threads gelegentlich auch als „lightweight processes“ bezeichnet. Von der Hierarchie her werden Threads zur Parallelisierung von von Abläufen innerhalb eines Prozesses eingesetzt, während der Prozess zur Parallelisierung auf Betriebssystemebene dient. Das folgende Bild einer „Programmierküche“ veranschaulicht die Zusammenhänge.



Während sich die Prozesse nur Ressourcen (den Speicher als Ganzes) auf Betriebssystemebene teilen und die ihnen zugeteilten Speicherbereiche (Adressräume) komplett getrennt sind (getrennte Küchen), teilen sich Threads (Köche) innerhalb eines Prozesses (Küche) Objekte bzw. den Speicherbereich.

Threads werden in vielen Programmiersprachen verwandt wie z.B. Delphi oder C/C++, dort muss man dazu meist eine spezielle (zusätzliche) Bibliothek zum Arbeiten mit Threads verwenden, häufig bietet auch das jeweilige Betriebssystem eine solche Bibliothek an.
Im Unterschied zu den eben genannten Bespielen sind Threads in Java bereits in den Sprachkern selbst integriert. Auch wenn man im eigenen Java-Programm nicht explizit mit Threads arbeitet, so gibt es kein normales Java-Programm, das nicht „multithreaded“ ist. So gibt es bei jedem Programm einen Thread, der die main - Methode des Programms (die statische main-Methode der gestarteten Klasse) ausführt und den Garbage-Collector- Thread, der immer im Hintergrund läuft. Verfügt das Programm über eine Gui, so kommt noch ein Event-Dispatcher-Thread hinzu, der die Gui-Ereignisse verarbeitet. Auch beim Laden von Bildern, bei den das Programm weiter arbeitet bevor das ganze Bild geladen ist, werden Threads verwendet (siehe auch MediaTracker, ImageObserver, Toolkit.getImage).

2. Ein konkretes Beispiel (Pi-berechnung, Gui-Reaktion):

Im Folgenden wird zuerst betrachtet wie man Threads erzeugt und startet und dann wird ein Beispiel betrachtet, in welchen Threads nützlich sind, um ein Programmierproblem zu beheben. Zur Erzeugung von Threads stellt Java die Klasse java.lang.Thread bereit. Um nun einen eigenen Thread (parallelen Programmablauf) zu bewerkstelligen sind 3 Dinge notwendig

1.) Eine eigene Klasse von Thread ableiten und in dieser die run-Methode überschreiben.
2.) Eine Instanz dieser Klasse erzeugen.
3.) Die start-Methode dieser Instanz aufrufen.

Dahinter steckt der folgende Mechanismus, wenn von einem Thread-Objekt (Instanz von Thread oder einer von Thread abgeleiteten Klasse), die start-Methode aufruft, so sorgt diese dafür, dass der in ihrer run-Methode stehende Programmcode nun parallel zum restlichen Programm ausgeführt wird.

```
...
class MyThread extends java.lang.Thread //Punkt1
{
 ...
      public void run()
      {
            ….irgendwas parallel Auszuführendes….
      }
}
…
MyThread mt = new MyThread();       //Punkt 2
mt.start();                         //Punkt 3
…
```

Man beachte, dass ein Thread (und damit die parallele Programmausführung) beendet ist, wenn er das Ende seiner run-Methode erreicht hat. Das Thread-Objekt selber existiert noch, allerdings kann den Thread nicht erneut starten, indem man die start-Methode einfach erneut aufruft. Möchte man den Thread erneut starten, so muss man erst ein neues Thread-Objekt (Instanz) erzeugen und von dieser die start-Methode aufrufen. Die Thread-Objekte beendeter Threads werden dann wieder andere Objekte auch behandelt und vom Garbage-Collector gelöscht, sobald keine Referenzen mehr auf sie zeigen.

Nun aber zu einem praktischen Programmierproblem. Das Programm PiBerechnung berechnet Pi nach dem Monte-Carlo-Verfahren, diese Berechnung kann sehr langwierig sein, wenn man eine hohe Punktezahl wählt. Der Benutzer wählt nun eine Punktzahl in der Gui un dieses Event wird nun in der folgendenen Methode bearbeitet:

```
 void jComboBox1_actionPerformed(ActionEvent e) {
    int n,i,inside;
    double x,y,pi;

    n = Integer.parseInt((String) jComboBox1.getSelectedItem());
    inside=0;
    for(i=0;i<n;i++) {
      x = Math.random();
      y = Math.random();
      if(y<Math.sqrt(1-x*x)) inside++;
      pi = 4.0*inside*1.0/i;
      jLabel2.setText("Pi = "+pi);
    }
  }
```

Die Variable n enthält die Anzahl der Punkte, die für die Berechnung verwendet werden sollen. Ist n sehr groß, so ist die Berechnung zeitaufwändig. Dies ist jedoch ein Problem, da das nächste Ereignis der Gui erst bearbeitet werden kann, wenn das derzeitige Ereignis (Berechnung von Pi) abgearbeitet ist. Dauert dies nun länger, so ist die Gui für diese Zeit blockiert und reagiert nicht mehr auf Benutzereingaben, der Benutzer kann das Programm auch nicht unterbrechen, da dementsprechende Ereignisse bzw. Reaktionen auf Benutzereingaben erst bearbeitet werden, nachdem die Berechnung von Pi zu Ende ist. Dies ist ein äußert ungünstiges Verhalten eines Programms. Eine mögliche Abhilfe besteht nun darin, die Bearbeitung der Gui-Ereignisse und der Berechnung von Pi parallel laufen zu lassen. Dazu erzeugt man nun in der obigen Methode einen Thread zur Berechnung von Pi, dieser läuft dann parallel zum restlichen Programm. Die Methode `jComboBox1_actionPerformed` dadurch nun schnell beendet werden und blockiert nicht mehr die Bearbeitung neuer Gui-Ereignisse.

```
void jComboBox1_actionPerformed(ActionEvent e) {
    if(currentThread!=null) {
      currentThread.stopThread();
      try {
        currentThread.join();
      } catch (Exception ex) {ex.printStackTrace();}
    }
    counter++;
    currentThread = new MyThread(""+counter);
    currentThread.start();
  }
}
```

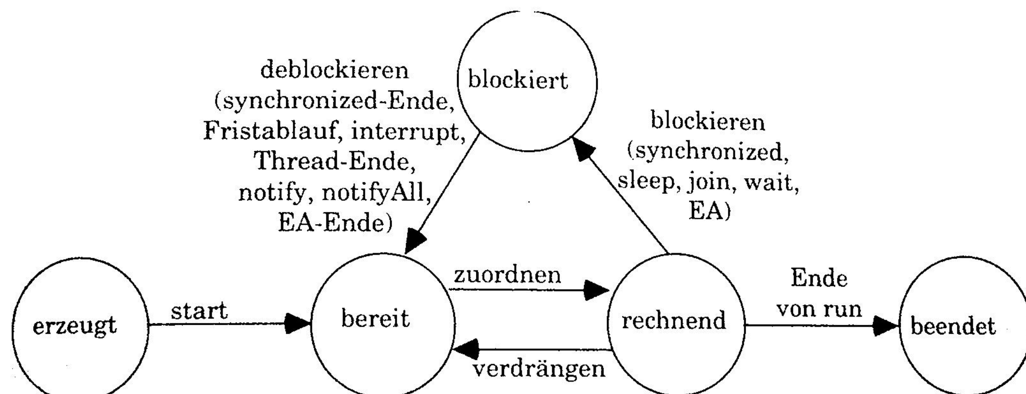
Die komplette Berechnung von Pi ist nun in die run-Methode der Klasse MyThread verlagert. In der Methode wird nur noch eine Instanz von MyThread erzeugt und gestartet. Die stopThread zeigt eine Möglichkeit einen Thread bzw. dessen run - Methode vorzeitig zu beenden. Man beachte hierbei, das der Thread im Normalfall noch direkt nach dem Aufruf der stoptThread- Methode beendet ist. Die Methode `currentThread.join()`hält den momentanen Thread solange an, bis der zu currentThread gehörige Thread beendet ist. So wird sicher gestellt, das immer nur ein Thread zur Berechnung von Pi gleichzeitig läuft.

```
class MyThread extends java.lang.Thread
{
```

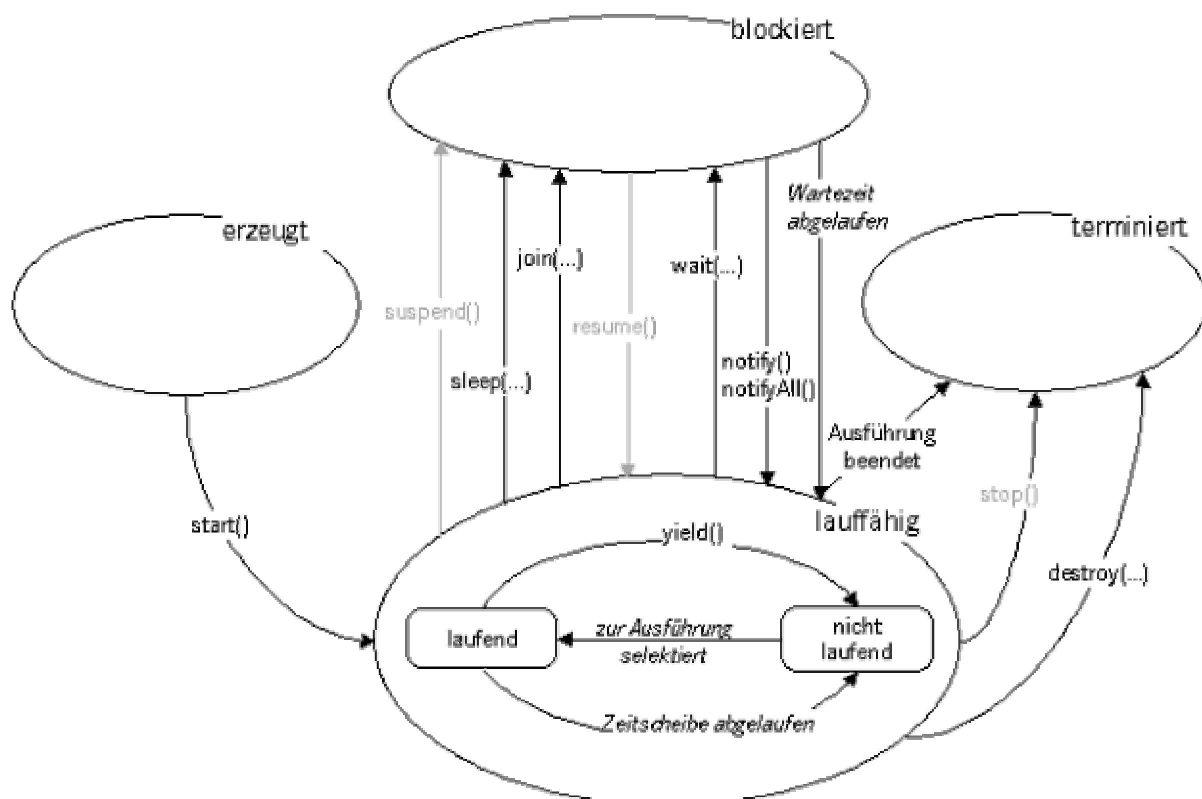
```
    public void run()
    {
      int n,i,inside;
      double x,y,pi;

      n = Integer.parseInt((String) jComboBox1.getSelectedItem());
      inside=0;
      i = 0;
      stop = false;
      while(i<n && !stop) {
        i++;
        x = Math.random();
        y = Math.random();
        if(y<Math.sqrt(1-x*x)) inside++;
        pi = 4.0*inside*1.0/i;
        jLabel2.setText("Pi = "+pi);
       }
    }

    public void stopThread()
    {
      stop = true;
    }
}
```

3. Thread-Zustände und Zeitscheibenverteilung.

Nun soll der Parallelismus von Threads genauer untersucht werden. Im Normalfall arbeiten Threads nicht wirklich parallel sondern simulieren dies nur. Während auf Rechnern mit nur einer CPU dies offentsichlich so sein muss, können bei Rechnern mit mehreren CPUs Threads auch wirklich parallel laufen, aber auch in diesen Szenario teilen sich meist mehrere Threads eine CPU, da man meist mehr Threads als CPUs hat. Die Parallelität wird von der JVM (Java Virtual Machine) durch ein so genanntes Zeitscheibenverfahren (time slicing) simuliert. Hierbei wird nacheinander jedem Thread für eine bestimmte Zeit (**Zeitscheibe**) Rechenzeit gewährt, dann wird er angehalten (verdrängt) und der nächste Thread kommt dran und so weiter bis man alle Threads durch hat und dann beginnt es wieder mit dem ersten. Ein Thread kann sich hierbei in 4 verschiedenen Zuständen befinden, die in der folgenden Grafik veranschaulicht sind.



Der Thread der im Moment Rechenzeit von der JVM bekommen hat befindet sich im Zustand *rechnend*, die anderen Threads denen die Rechenzeit entzogen worden ist befinden sich im Zustand *bereit*. Die Wechsel von *bereit* nach *rechnend* und umgekehrt wird im Normafall von der JVM im Zeitscheiben verfahren verursacht und ist außerhalb der Kontrolle des Programms. Der Zustand *blockiert* wird jedoch vom Programm selbst herbeigeführt, indem es einen Thread expliziert anhält, dies kann zum Beispiel durch im vorherigen Kapitel verwendete join-Methode geschehen. Weitere in der Zeichnung angebene Methoden werden später noch erläutert. Im Zustand *erzeugt* befindet sich ein Thread, wenn seine zugehörige Instanz erzeugt aber seine start-Methode noch nicht aufgerufen worden ist. Man beachte. Dass ein Thread auch nach dem Aufruf der start-Methode nicht direkt rechnet, sondern lediglich in den Zustand *bereit* wechselt und erst dann wirklich rechnet wenn die JVM ihm eine Zeitscheibe zuteilt und in den Zustand *rechnend* versetzt. Hat ein Thread das Ende seiner run-Methode erreicht so befindet er sich im Zustand *beendet* . Insgesamt sind die Zustände *erzeugt, blockiert* und *beendet* von der Zeitscheibenverteilung ausgeschlossen, die nur zwischen Threads in den Zuständen *bereit* und *rechnend* stattfindet. Allerdings kann man einen Thread von *erzeugt* und *blockiert* wieder nach *bereit* versetzen und ihn so wieder an der Zeitscheibenverteilung teilnehmen lassen. Im Zustand *beendet* ist ein Thread jedoch endültig ausgeschieden und kann nicht mehr verwendet werden. In der zweiten Grafik werden die verschiedenen zustände noch einmal mit größerer Detailinformation dargestellt, insbesondere kann man sehen mit welchen Methoden ein Programm die Zustände beeinflussen kann. Man beachte hier, dass die grau dargestellten Methoden (stop, suspend, resume) Sun als „deprecated“ deklariert wurden und daher nicht mehr verwendet werden sollten. Der Grund dafür ist, das ihre Verwendung unter Umständen zu Verklemmungen (siehe späteres Kapitel) und Inkonsistenzen führen kann.



4. Thread-Api

Die folgenden Klassen/Methoden werden von Threads verwandt, dies ist keine vollständige sondern eine exemplarische Beschreibung der wichtigsten Klassen, Methoden, Mechanismen zur Thread-Programmierung. Eine vollständige Beschreibung findet man unter [1] und [5]. Neben der schon bekannten Klasse Thread sind vor allem das synchronized Schlüsselwort und die wait- und notify Methoden von großer Bedeutung. Mit synchronized kann man sicherstellen, dass bestimmte Programmteile oder Objekte nicht gleichzeitig von mehreren Threads verwendet werden können. Es kann zu jedem Zeitpunkt nur maximal ein Thread auf sie zugreifen. Die Methoden wait und notify werden verwendet, um Threads zu blockieren (passives Warten) und zu entblockieren. Soll ein Thread zum Beispiel auf ein bestimmtes Ereignis warten, so lässt man ihn nicht ständig in eine Schleife selber nach dem Ereignis fragen (aktives Warten), sondern blockiert ihn zuerst mit wait() und entblockiert ihn dann später wenn das Ereignis eintritt mit notify (passives Warten). Auf diese Weise ist warten effizient und man verschwendet keine Rechenzeit, siehe hierzu auch die Timer-Implementierung.

```
class Beispiel
{
      public synchronized void methode1() {…}
      public void methode2()
      {
            synchronized(objReference)
            {
                  ……
            }
      }
}

public interface Runnable
{
      public void run();
}

public class Thread
{
      public Thread()
      public Thread(Runnable r)
      public void run();
      public void start()
      public final Boolean isAlive()
      public final void join()
      public static sleep(long milliseconds)
}

public class Object
{
      public final void wait();
      public final void wait(long milliseconds);
      public final void notify();
      public final void notifyAll();
}
```

5. Timer

Timer bieten die Möglichkeit einen Programmcode zu einem bestimmten Zeitpunkt auszuführen oder ihn auch periodisch in einem bestimmten Zeitintervall ausführen zu lassen. Somit Timer stellen eine weitere Lösung unseres Ausgangsproblem mit der Pi-Berechnung dar und sie lassen sich außerdem gut mit Hilfe von Threads implementieren. Bei der Pi-Berechnung kann man auch anstatt die ganze langwierige Berechnung in einen Thread abzuschieben, versuchen die Berechnung in kleinere Schritte zu zerlegen. Man hat dann jedoch das Problem dafür zu sorgen, dass diese alle hintereinander ausgeführt werden. Die Methode `jComboBox1_actionPerformed` wird ja als Reaktion auf die Benutzereingabe nur einmal aufgerufen, kann also höchstens einen dieser Schritte ausführen. Man kann aber den Timer-Mechanismus nutzen, um dafür zu sorgen, dass alle Schritte ausgeführt werden. Man lässt periodisch eine Methode ausführen, die einen einzelnen Rechenschritt darstellt Bevor wird dies jedoch im Detail betrachtet wird, soll noch etwas zur Verwendung von Timern in Java gesagt werden. Java stellt hierzu 2 Klassen **Timer** und **TimerTask** zur Verfügung. Die run-Methode von TimerTask enthält den Programmcode, der zu einem bestimmten Zeitpunkt ausgeführt werden soll. Aslo muss man ganz analog zum Verfahren bei Threads die Klasse TimerTask ableiten und deren run-Methode überschreiben. Die Klasse Timer enthält eine Methode, um den Ausführungszeitpunkt der run-Methode von TimerTask festzulegen. Insgesamt muss man folgende Schritte durchführen.

1.) Die Klasse TimterTask ableiten und deren run-Methode überschreiben.
2.) Eine Instanz A dieser Klasse erzeugen.
3.) Eine Instanz B der Klasse Timer erzeugen
4.) Die schedule-Methode von B verwenden, um den Ausführungszeitpunkt der run-Method von A festzulegen.

```
…
class MyTimerTask extends java.util.TimerTask
{
…
      public void run()
      {
            auszuführender Programmcode
      }
…
}
…
Timer ti = new Timer();
MyTimerTask task = new MyTimterTask();
Ti.schedule(task,startzeit,periode);
….
```

Die Pi-Berechnung lässt sich damit nun wie folgt umschreiben:

```
  void jComboBox1_actionPerformed(ActionEvent e) {
    int  n = Integer.parseInt((String) jComboBox1.getSelectedItem());
    if(currentTimerTask!=null) currentTimerTask.cancel();  //<== Änderung !
    counter++;
    currentTimerTask = new MyTimerTask(""+counter,n); // <== Änderung !
    myTimer.schedule(currentTimerTask,0,1000);        // <== Änderung !
  }
```

Als Reaktion auf die Benutzereingabe wird nun kein Thread erzeugt sondern ein TimerTask und dieser wird dann gleich mit der schedule-Methode zur periodischen Ausführung vorgesehen. Die eigentliche Pi-Berechnung findet nun in der run-Methode von MyTimerTask statt, allerdings befindet sich dort jeweils nur ein Berechnungsschritt und nicht die komplette Berechnung, diese setzt sich aus den einzelnen Aufrufen zusammen. Man beachte auch dass

einige lokale Variablen (n, inside, current) von zu Membervariablen der Klasse MyTimterTask geworden sind, damit die Zwischenergebnisse der einzelnen Rechenschritte nicht verloren gehen. Die cancel-Methode von TimerTask sorgt dafür, dass die run-Methode nicht mehr ausgeführt wird, so kann man den periodischen Aufruf abbrechen.

```
class MyTimerTask extends java.util.TimerTask
  {
    final int step = 10; // <== Änderung !
    // Timeraufruf nicht zurückgesetzt werden sollen.
    int inside = 0;     // <== Änderung !
    int n;              // <== Änderung !
    int current = 0;    // <== Änderung !

    public MyTimerTask(String id, int n)
    {
      this.id = id;
      this.n = n;
    }

    public void run()
    {
      int i,max;
      double x,y,pi;
      if(current < n) {
        for (i = 0; i < step; i++) {
          x = Math.random();
          y = Math.random();
          if (y < Math.sqrt(1 - x * x)) inside++;
        }
        current = current + step;
        pi = 4.0 * inside * 1.0 / current;
        jLabel2.setText("Pi = " + pi);
      } else {
        this.cancel();
      }
    }
  }
```

6. Thread synchronization und Timer-Implementation

In vielen Fällen arbeiteten Threads nicht wirklich unabhängig voneinander vor sich hin, sondern beeinflussen sich gegenseitig. So wartet ein Thread zum Beispiel auf das Ergebnis eines anderen oder mehrere Threads müssen sich ein synchronized Objekt teilen oder Ähnliches. Im folgenden wird noch einmal kurz auf die Mechanismen eingegangen die Java zur Verfügung stellt zum Threads zu synchronisieren und so bestimmte Abläufe und Anordnungen zu erzwingen. In der Literatur findet man zu dieser Thematik meist die Begriffe Semaphor und Mutex und innerhalb der Windows-Programmierung auch den Begriff Critical Section. Java kennt diese nicht und verwendet stattdessen synchronized, wait und notify. Diese klassischen Konzepte lassen sich aber mit Hilfe der Java-Mechanismen problemlos nachprogrammieren, die meisten Probleme lassen sich aber mit ihnen auch einfach direkt ohne diesen Umweg lösen. Die hier verwendeten Beispiele verwenden ausschließlich die Java-Mechanismen.

synchronized:

Mit diesem Schlüsselwort erzeugt man eine Sperre (lock) auf ein Objekt, dieses kann nur von demjenigen Thread benutzt werden, der die Sperre besitzt. Alle anderen Threads müssen warten bis die Sperre wieder freigegeben wird. Hierbei kann man eine Methode synchronized deklarieren oder einen Block. Ruft ein Thread eine synchronized Methode eines Objektes auf, so erhält er die Sperre auf dieses Objekt, wenn sie noch erhältlich ist oder er muss andernfalls warten bis sie erhältlich wird. Besitzt er die Sperre kann nur er allein die synchronized Methoden dieses Objektes verwenden bis er die Sperre wieder abgibt. Die Sperre wird wieder abgegeben, wenn der Thread die synchronized Methode oder den synchronized Block verlässt.

```
…
synchronized(object) // Sperre von Objekt besorgen
{
   Programmcode wird nur von maximal einem Thread "gleichzeitig" ausgeführt
}
…
class Beispiel
{
      public synchronized void methode1a() {…}
      public void methode1b() // alternative Definition von methode1a
      {
            synchronized(this)
            {
                  ……
            }
      }
      public synchronized void methode2() {…}
      public void methode3(){…}// kann von anderen Threads benutzt werden

}

Beispiel  b = new Beispiel()
b.methode1a()
```

Solange der Thread die methode1a abarbeitet kann kein anderer Thread eine synchronized Methode von b benutzen oder einen Block ausführen der mit b als Objekt synchronized ist.

wait und notify

Diese beide Methoden dienen zum Blockieren und Entblockieren eines Threads. Sie können nur in innerhalb eines synchronized Block oder einer synchronized Methode verwandt werden. Die wait-Methode blockiert einen Thread bis er mit der notify-Methode wieder entblockiert wird oder ein bestimmtes Zeitintervall verstrichen ist. Das ist zu beachten, dass beim Aufruf der wait-Methode die vorher mit synchronized erlangte Sperre wieder frei gegeben wird. In diesem Fall also bevor der Thread den synchronized Block wieder verlassen hat. Wenn der Thread später die Programmausführung fortsetzt, so muss er sich zuerst wieder die Sperre besorgen und gegebenenfalls erst warten bis sie von anderen Threads freigegeben wird (er befindet sich solange noch im Zustand *blockiert*). Hat er die Sperre wieder bekommen, so setzt er die Ausführung fort und gibt die Sperre wieder frei, wenn der den synchronized block verlässt. Während die notify-Methode nur einen einzelnen mit wait blockierten Thread entblockiert, entblockiert die notifyAll-Methode alle an diesem Objekt dirch wait blockierten Threads. Dieser Unterschied kann sehr wichtig sein, da in manchen Situationen eventuell nur einer der blockierten Threads weiterarbeiten kann. Die notify-Methode entblockiert jedoch nur einen Thread und man hat keinen Einfluss darauf welcher dies ist. Daher besteht die Möglichkeit , dass man den falschen aufgeweckt hat und der

einzige der weiterarbeiten könnte weiterhin blockiert ist. Mit der notifyAll-Methode lässt sich dieser unschöne Fall verhindern.

Timer-Implementierung:

Wir orientieren uns an der Java-Api und implementieren eine eigene Timer und TimerTask Klasse. Hierbei soll Timer einen nach Ausführungszeit sortierte Liste von TimerTask-Instanzen verfügen und überprüft in einer Schleife ständig, ob die Ausführungszeit für eine der TimerTask-Instanzen erreicht ist. Ist dies der Fall so ruft er dessen run-Methode auf. Damit all dies parallel/unabhänging zum restlichen Programm geschieht, erzeugt die Timer Klasse einen eigenen Thread in dem diese Schleife läuft.

Das folgende Listing enhält die wichtigsten Elemente der Timer Klasse. Sie ist von Thread abgeleitet und starten sich in ihrem Konstruktor selbst (19-24). Die innere Klasse ScheduledTask (3-15) dient nur dazu, die in der schedule-Methode übergeben TimerTask Instanzen mit ihrer Ausführungszeit zu koppeln. Die Membervariable scheduledTasks (17) enthält, dann die nach ihrer Ausführungszeit sortiertenTimterTask Instanzen. Die Schleife in der ständig die TimerTask Instanzen überprüft werden entspricht der while-Schfeife (44-74). Hier wird dann zuerst die Liste durchgegangen und zuuerst geprüft ob die TimerTask Instanz noch nicht gecancelt worden ist (49) . Dann werden alle Instanzen deren Ausführungszeit erreicht ausgeführt, dann werden sie aus der Liste entfernt und mit ihrer neuen Ausführungszeit wieder eingetragen (50-57). Wird die erste TimerTask Instanz, die noch nicht ausgeführt werden muss erreicht, so wird die Wartezeit berechnet, die der Thread blockiert sein kann, um sie noch rechzeitig auszuführen(59-60). Danach wird der Thread für die entsprechende Zeit blockiert (69-73). Wird nun aber in der Zwischenzeit während der Thread blockiert ist eine neue TimerTask Instanz mit der schedule-Methode hinzugefügt, dann könnte deren Ausführungszeit ja vor dem Entblockierungszeitpunkt liegen, deshalb entblockiert die schedule-Methode (34-36) den Thread.

```
1   public class Timer extends Thread {
2
3     class ScheduledTask
4     {
5       TimerTask timerTask;
6       long executionTime;
7       long period;
8
9       ScheduledTask(TimerTask timerTask, long executionTime, long period)
10      {
11        this.timerTask = timerTask;
12        this.executionTime  = executionTime;
13        this.period = period;
14      }
15    }
16
17    Vector scheduledTasks;
18
19    public Timer()
20    {
21      this.active=true;
22      this.scheduledTasks = new Vector();
23      this.start();
24    }
25
26
27    public void schedule(TimerTask timerTask, long start, long period)
28    {
29      int i,max;
30      max = scheduledTasks.size();
31      ScheduledTask scheduledTask = new ScheduledTask(timerTask,
32                                 System.currentTimeMillis()+start,period);
33      insertByTime(scheduledTask);
34      synchronized(this) {
35          this.notify();
36      }
37    }
38
39    public void run()
40    {
41      int i,max;
42      ScheduledTask scheduledTask;
43      long currentTime,waitTime;
44      while(active) {
45        waitTime = Long.MAX_VALUE;
46        max = scheduledTasks.size();
47        for(i=0;i<max;i++) {
48          scheduledTask = (ScheduledTask) scheduledTasks.elementAt(i);
49          if(scheduledTask.timerTask.isActive()) {
50            currentTime = System.currentTimeMillis();
51            if(scheduledTask.executionTime <= currentTime) {
52               scheduledTask.timerTask.run();
53               scheduledTasks.remove(scheduledTask);
54               scheduledTask.executionTime =   currentTime
55                                             + scheduledTask.period;
56               insertByTime(scheduledTask);
57               i--;
58            } else {
59              waitTime =    scheduledTask.executionTime
60                          - System.currentTimeMillis();
61              i=max;
62            }
63          } else {
```

```
64              scheduledTasks.remove(scheduledTask);
65              max = scheduledTasks.size();
66              i--;
67            }
68          } //for
69          synchronized (this) {
70            try {
71               this.wait(waitTime);
72            } catch (Exception e) {}
73          }
74        }//while
75      }
76    }
```

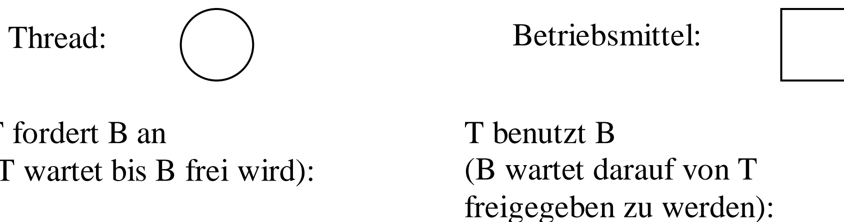
7. Deadlocks/Verklemmungen

Verklemmungen sind eine klassische Fehlerquelle bei Programmierung von Threads, sie sind häufig nur schwer aufzudecken oder zu debuggen, da sie sich unter Umständen nur schwer reproduzieren lassen. Doch bevor wir weiter in die Problematik einsteigen, sollen zuerst einige grundlegende Begriffe geklärt werden.

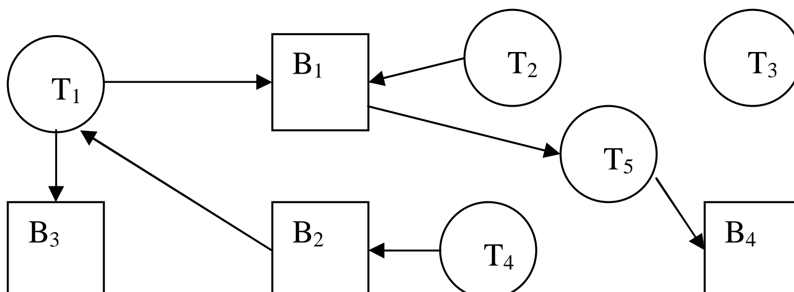
Definition: Betriebsmittel:
Ein **Betriebsmittel** (engl. **resource**) ist ein Objekt, auf das ein Thread unter Umständen warten muss, bevor er es benutzen kann. In Java bedeutet dies, dass der Zugriff auf das Objekt über synchronized-Methoden oder Methoden, die synchronized-Blöcke erfolgt.

Definition: Verklemmung:
Eine Menge von Threads befindet sich in einer **Verklemmung** (engl. **deadlock**), falls jeder Thread dieser Menge auf ein Ereignis wartet, dass nur ein anderer Thread dieser Menge auslösen kann. Dies bedeutet die Threads blockieren sich gegenseitig und das Programm steht still.

Ein **Betriebsmittelgraph** bietet stellt graphisch dar, welche Threads welche Betriebsmittel zu einen bestimmten Zeitpunkt benutzen. Man benutzt hierbei folgende Notation:



Ein Beispiel:

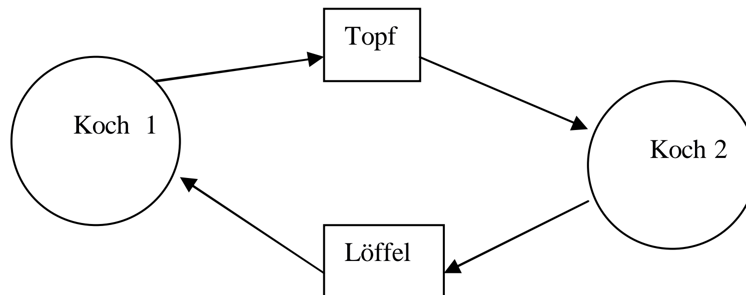


Beispiel 1: Blockierte Köche (oder zu viele Köche verderben den Brei):

Es gibt 2 Köche (Threads), die gleichzeitig kochen. Hierbei benötigt jeder zum Kochen eine Topf und einen Löffel. Topf und Löffel können zu einem Zeitpunkt nur von höchstens einem

Koch benutzt werden. Wenn einer von ihnen von einem Koch benutzt wird, so muss der andere Koch warten bis sie wieder zurück ins Regal gestellt werden (freigegeben). Betrachte nun folgende Situation: Koch 1 nimmt den Löffel und Koch 2 den Topf. Nun versucht Koch 1 auch den Topf zu nehmen und muss warten, Koch 2 wiederum möchte den nun den Löffel nehmen und muss nun ebenfalls warten. Was nun ?

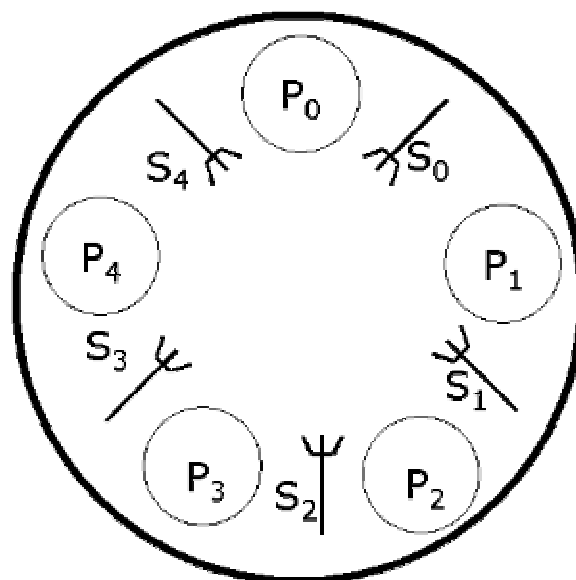
Offensichtlich blockieren sich hier beide Köche gegenseitig und man hat eine Verklemmung erzeugt. Betrachtet man den zugehörigen Betriebsmittelgraphen so stellt man fest. Dass er einen Zyklus besitzt.



In der Tat liegt eine Verklemmung genau dann vor, wenn der momentane Betriebsmittelgraph einen Zyklus besitzt.

Beispiel 2: Die Hungrigen Philosophen (dining philosophers):

Um einen Tisch sitzen n Philosophen, die abwechselnd (zufällig) essen und denken. Jeder Philosoph hat vor sich einen Teller und rechts und links von sich eine Gabel, wobei es insgesamt n Teller und n Gabeln gibt (siehe Zeichnung). Ein Philosoph benötigt zum Essen 2 Gabeln (recht und links von ihm), dies bedeutet das keine 2 benachbarten Philosophen gleichzeitig essen können. Wenn ein Philosoph essen möchte, muss er warten bis die Gabeln rechts und links von ihm frei sind. Die Zeichnung illustriert den Fall n=5.



Dies ist wohl eines der bekanntesten Beispiele aus der Literatur und auch hier kann es zu einem Verklemmungsszenario kommen. Angenommen alle Philosophen ergreifen mehr oder weniger gleichzeitig die Gabel links von ihnen, nachdem sie sich entschieden haben zu essen

und versuchen dann die Gabel rechts von ihnen aufzunehmen. Diese sind jedoch jeweils schon in der linken Hand ihre rechten Nachbarn, also warten nun alle (auf Godot) darauf die 2-te Gabel zu bekommen und es liegt auch eine Verklemmung vor. Der zugehörige Betriebsmittelgraph zeigt auch wieder einen Zyklus. Betrachtet man die die Philosophen für den Fall n = 2, so erhält man das Bespiel mit den Köchen.

Man kann solche Verklemmungsszenarien vermeiden, indem man entweder alle Betriebsmittel auf einen Schlag anfordert, d.h. ein Koch nimmt Topf und Löffel gleichzeitig oder Philosoph beide Gabeln. Eine weitere Strategie ist, dass man verlangt, dass Betriebsmittel immer nur in einer bestimmten Reihenfolge angefordert werden dürfen. Würden z.B. im Kochbeispiel beide Köche immer zuerst den Topf und dann den Löffel nehmen, so wäre auch keine Verklemmung möglich. Es gewinnt sozusagen derjenige der Topf zuerst hat und der andere wartet darauf, dass er ohne zurückgestellt wird ohne in der Zwischenzeit zu versuchen den Löffel zu bekommen.

Zum Abschluss sein nun noch ein Ausschnitt aus einer Implementation des Kochbeispiels angegeben.

Die beiden Köche sind hierbei Instanzen der Klasse Koch und Topf und Löffeö sind Instanzen der Klasse Betrieb. Wichtig ist das in der run-Methode der Klasse Koche die Betriebsmittel einzeln und nicht auf einen Schlag angefordert werden (18-32), weiterhin kann mit der auskommentierten Zeile 23, die Verklemmung sozusagen gleich zu Beginn erzwingen, da durch die yield-Methode der Thread sofort in den Zustand *bereit* versetzt wird, das der andere Koch zum Zug kommt. Wichtig ist auch noch das den beiden Köchen die Betriebsmittel in verschiedener Reihenfolge übergeben werden (77-78), so dass der eine Koche zuerst zum Löffel greift und der andere zuerst nach dem Topf, denn nur so lässt sich die Verklemmung erreichen.

```
1   class Koch extends Thread
2   {
3     KuechenBetriebsmittel kbm1,kbm2;
4     boolean cooking;
5     JLabel label;
6     long counter;
7
8     public Koch(String name,JLabel label,KuechenBetriebsmittel kbm1,
9                KuechenBetriebsmittel kbm2)
10    {
11      super(name);
12      this.kbm1 = kbm1;
13      this.kbm2 = kbm2;
14      this.label = label;
15      this.cooking=true;
16    }
17
18    public void run()
19    {
20      counter=0;
21      while(cooking) {
22        kbm1.holen(this.getName());
23        //if(Math.random()< 0.25) this.yield();
24        kbm2.holen(this.getName());
25        counter++;
26        label.setText(this.getName()+" kocht Mahlzeit "+counter);
27        try{ Thread.sleep(100);} catch(Exception ex) {};
28        kbm1.zurueckstellen(this.getName());
29        kbm2.zurueckstellen(this.getName());
30        label.setText(this.getName()+" ist fertig mit Mahlzeit "+counter
31                                    +" und macht eine kleine Pause");
32        try{ Thread.sleep(5);} catch(Exception ex) {};
33      }
34    }
35  }
36
37  class KuechenBetriebsmittel
38  {
39    boolean used;
40    String name,user="niemand";
41    JLabel label;
42
43    public KuechenBetriebsmittel(String name,JLabel label)
44    {
45      this.name = name;
46      this.used = false;
47      this.label = label;
48    }
49
50    public synchronized void holen(String str)
51    {
52      try {
53        label.setText(str + " versucht "+this.name+" zu holen und "
54                        +this.name+" wird zur Zeit von "+this.user
55                        + " benutzt");
56        if (used) this.wait();
57        this.used = true;
58        this.user = str;
59      } catch(Exception ex) {ex.printStackTrace();}
60    }
61    public synchronized void zurueckstellen(String str)
62    {
63      this.notify();
```

```
64       this.used = false;
65       this.user = "niemand";
66       label.setText(str + " stellt "+this.name+" zurück und "
67                     +this.name+" wird zur Zeit von "+this.user+ " benutzt");
68      }
69   }
70
71   …
72     void jButtonStart_actionPerformed(ActionEvent e) {
73       KuechenBetriebsmittel topf = new KuechenBetriebsmittel("Topf",
74                                                    this.jLabel5);
75       KuechenBetriebsmittel loeffel = new KuechenBetriebsmittel("Löffel",
76                                                          this.jLabel6);
77       Koch koch1 = new Koch("Koch1",this.jLabel2 ,topf,loeffel);
78       Koch koch2 = new Koch("Koch2",this.jLabel3 ,loeffel,topf);
79       koch1.start();
80       koch2.start();
81     }
82   …
```

8. Programme

Die folgenden Programme liegen als JBuilder-Projekte vor, unter anderem sind alle im Skript verwandten Programmteile in ihnen enthalten und meist noch zusätzlichen Kommentaren versehen.

PiBerechnung
PiBerechnungThreads
PiBerechnungTimer
PiBerechnungMPGTimer
Veklemmung

9. Literatur

[1] Java Api Dokumentation (Sun):
http://java.sun.com/j2se/1.4.2/docs/api/

[2] Java Tutorial (Sun) zu Threads:
http://java.sun.com/docs/books/tutorial/essential/threads/index.html

[3] Bruce Eckel : Thinking in Java (Kapitel zu Threads)
http://mindview.net/Books

[4] Guido Krüger : Handbuch der Javaprogrammierung (Kapitel zu Threads)
http://www.javabuch.de/

[5] Vorlesung des FH Augsburg zu Threads in Java:
http://www.jeckle.de/vorlesung/javaThreads/script.html

[6] Jeff Friesen: Achieve strong performance with threads (Java World)
http://www.javaworld.com/javaworld/jw-05-2002/jw-0503-java101.html

[7] Allen Holub: Programming Java threads in the real world (Java World)
http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads.html

[8] Threads in Delphi:
http://www.pergolesi.demon.co.uk/prog/threads/ToC.html