Threads

in Java

by

Dr. Günter Kolousek

Threads

- Grundbausteine paralleler Software
- aus Sicht des Betriebssystems: kleinste Einheit der Parallelität
- ein Prozess besteht aus 1 oder mehreren Threads
- alle Threads innerhalb eines Prozesses können auf alle Ressourcen des Prozesses zugreifen
- jeder Thread hat separat Stack, Registerinhalte, Schedulingparameter (Priorität, Affinität,…). Außerdem: thread-lokale Daten

Prozess vs. Thread

- Prozess
  - Vorteile
    - Nichtbeinflussung anderer Prozesse
    - Rechte, Abrechnung
  - Nachteile
    - Anlegen ressourcenintensiv (Zeit, Speicher)
    - Context Switch zeitintensiv: CPU Kontext (Register, Programmzähler, Stackpointer,…), MMU Register, Swapping, CPU Abrechnungen,…
- Thread
  - Vorteile
    - geringerer Overhead beim Anlegen & Context Switch
    - Zugriff auf Daten und offene Dateien,…
  - Nachteile
    - Beeinflussung durch andere Threads

Einsatz auf Single-Core Systemen

- Asynchrones Warten (aus globaler Sicht)
  - Überbrückung der Wartezeit bei Ein- aber auch Ausgaben
- Responsiviät der Benutzeroberfläche
  - Bedienbarkeit trotz ”rechenintensiver” Applikation
- Trennung der Teilaufgaben
  - Aufsplittung von unabhängigen Aktivitäten
  - z.B. Musik, Kommunikation, Darstellung,… in einem Computerspiel

Starten von Threads

```
public class MyThread extends Thread {
    public void run() {
        int i=0;
        while (i < 10) {
            System.out.print(”A”);
            i++;
        } }
    public static void main(String[] args) {
        MyThread t=new MyThread();
        t.start();
        int i=0;
        while (i < 10) {
            System.out.print(”B”);
            i++;
        } } }
```

```
BBBBBBBBBBAAAAAAAAAA
```

Starten von Threads

```
public class MyThread extends Thread {
    public void run() {
        int i=0;
        while (i < 10) {
            System.out.print(”A”);
            i++;
        } }
    public static void main(String[] args) {
        MyThread t=new MyThread();
        t.start();
        int i=0;
        while (i < 10) {
            System.out.print(”B”);
            i++;
        } } }
```

BBBBBBBBBBAAAAAAAAAA

kein context switch?

Um einen context switch bitten

```
public class MyThread2 extends Thread {
    public void run() {
        int i=0;
        while (i < 10) {
            System.out.print(”A”);
            yield();
            i++;
        } }
    public static void main(String[] args) {
        MyThread2 t=new MyThread2();
        t.start();
        int i=0;
        while (i < 10) {
            System.out.print(”B”);
            yield();
            i++;
        } } }
```

mögliche Ausgabe: BABBBBBBABABABAAAAAA

Starten von Threads – 2

- immer von `Thread` ableiten?
  - Was wenn Klasse schon von anderer abgeleitet?
  - $\rightarrow$ Interface `Runnable`!

```
public class MyThread3 implements Runnable {
    public void run() {
        int i=0;
        while (i < 10) {  System.out.print(”A”);
            Thread.yield();  // static!
            i++;
        } }
    public static void main(String[] args) {
        MyThread3 t=new MyThread3();
        new Thread(t).start();
        int i=0;
        while (i < 10) {  System.out.print(”B”);
            Thread.yield();
            i++;
        } } }
```

Java-Prozess

- besteht aus mindestens einem Thread
  - `Thread.currentThread()`
- beendet sich wenn
  - `System.exit(n)` aufgerufen
    - äquivalent zu `Runtime.getRuntime().exit(n)`
  - *alle* (nicht Dämon-)Threads sich beendet haben
- neuer Prozess kann mit new `ProcessBuilder(...).start()` erzeugt (und gestartet) werden

Thread-Zustände

- `NEW` … neu angelegter Thread
- `RUNNABLE` … wird von JVM ausgeführt
- `BLOCKED` … wartet auf Lock
- `WAITING` … wartet auf auf anderen Thread
- `TIMED_WAITING` … wartet mit Zeitablauf
- `TERMINATED` … hat sich beendet

Warten bis anderer Thread endet

```
public class MyThread4 implements Runnable {
    public void run() {
        int i=0;
        while (i < 10) {
            System.out.print(”A”); Thread.yield(); i++;
        } }
    public static void main(String[] args) {
        MyThread4 my=new MyThread4();
        Thread t=new Thread(my);
        t.start();
        try { t.join(); } catch (InterruptedException e) {}
        int i=0;
        while (i < 10) {
            System.out.print(”B”);
            Thread.yield();
            i++;
        } } }
```

```
AAAAAAAAAABBBBBBBBBB
```

Threads und gemeinsame Daten

- Ein (Haupt)Vorteil von Threads ist…

Threads und gemeinsame Daten

- Ein (Haupt)Vorteil von Threads ist…
  Zugriff auf gemeinsame Daten

Threads und gemeinsame Daten

- Ein (Haupt)Vorteil von Threads ist…
  Zugriff auf gemeinsame Daten
- Regelung des Zugriffes auf gemeinsame Ressourcen von kritischen Abschnitten (engl. critical sections)

Threads und gemeinsame Daten

- Ein (Haupt)Vorteil von Threads ist…
  Zugriff auf gemeinsame Daten
- Regelung des Zugriffes auf gemeinsame Ressourcen von kritischen Abschnitten (engl. critical sections)
  - $\rightarrow$ Race conditions (…Wettkampfbedingung, Gleichzeitigkeitsbedingung)
  - $\rightarrow$ Synchronisation um wechselseitigen Ausschluss (engl. mutual exclusion) zu erreichen
  - $\rightarrow$ zustandsabhängige Steuerung (später)

Problematik

```
public class Account extends Thread {
    public int balance=15;

    boolean withdraw(int amount) {
        if (balance >= amount) {
          balance -= amount; return true;
        } else { return false; }
    }

    public void run() {
        System.out.print(withdraw(10) + " ");
    }

    public static void main(String[] args)
      throws InterruptedException {
        Account acc = new Account();  acc.start();
        System.out.print(acc.withdraw(6) + " ");
        acc.join();
        System.out.println(acc.balance);
    } }
```

Problematik – 2

Erwartetes Ergebnisse wären:

- `true false 9` oder
- `true false 5`

Problematik – 2

Erwartetes Ergebnisse wären:

- `true false 9` oder
- `true false 5`
- allerdings geht auch: `true true -1`

Problematik – 2

Erwartetes Ergebnisse wären:

- `true false 9` oder
- `true false 5`
- allerdings geht auch: `true true -1`
- oder z.B. auch: `true true 9` (!)

Problematik – 3

t1	t2	balance
`15 >= 10`		15
`balance - 10`		15
	`15 > 6`	15
	`balance - 6`	15
`balance = 5`		5
	`balance = 9`	9
	`success = true`	
`success = true`		

Lösung

```
public class SafeAccount extends Thread {
    public int balance=15;

    synchronized boolean withdraw(int amount) {
        if (balance >= amount) {
          balance -= amount; return true;
        } else { return false; } }

    public void run() {
        System.out.print(withdraw(10) + " "); }

    public static void main(String[] args)
      throws InterruptedException {
        SafeAccount acc = new SafeAccount();  acc.start();
        System.out.print(acc.withdraw(6) + " ");
        synchronized (acc) {
            System.out.println(acc.balance); } }
        }
        acc.join();
        System.out.println(acc.balance); } }
```

Race Conditions

- Arten
  - Write/Write

Race Conditions

- Arten
  - Write/Write
  - Read/Write

Race Conditions

- Arten
  - Write/Write
  - Read/Write
  - Read/Read

Race Conditions

- Arten
  - Write/Write
  - Read/Write
  - Read/Read **nein!**
- Mehrere Threads greifen zumindest schreibend auf gemeinsame Ressource zu

Race Conditions – 2

- Write/Write
  mind. 2 Threads schreiben

```
void double() {
    x = x * 2;  // write
}

void halve() {
    x = x / 2;  // write
}
```

Race Conditions – 3

- Read/Write
  ein Thread liest, einer schreibt

```
void calc_sides(double r, double phi) {
    a = r * Math.sin(phi);  // write
    b = r * Math.cos(phi);
}

void calc_area() {
    A = (a * b) / 2;  // read
}
```

Lösung

- Erreichung eines wechselseitigen Ausschlusses (engl. mutual exclusion)
- durch Synchronisation
- verschiedene Synchronisationsmechanismen existieren
- in Java wird hauptsächlich `synchronized` verwendet

Begriffe

- Synchronisation

  Synchronisation (griechisch: syn ≡ „zusammen“, chrónos ≡ „Zeit“) bezeichnet das zeitliche Aufeinander-Abstimmen von Vorgängen, Uhren und Zeitgebern. Synchronisation sorgt dafür, dass Vorgänge gleichzeitig (synchron) oder in einer bestimmten Reihenfolge ablaufen. *Wikipedia*

Begriffe

- Synchronisation

  > *Synchronisation (griechisch: syn ≡ „zusammen“, chrónos ≡ „Zeit“) bezeichnet das zeitliche Aufeinander-Abstimmen von Vorgängen, Uhren und Zeitgebern. Synchronisation sorgt dafür, dass Vorgänge gleichzeitig (synchron) oder in einer bestimmten Reihenfolge ablaufen.* *Wikipedia*

  Synchronisation beschreibt ein Verfahren wie Prozesse oder Threads sich untereinander abstimmen, um Aktionen in einer bestimmten Reihenfolge auszuführen.

Begriffe – 2

- Betriebsmittel, Ressource (engl. resource): Speicher, Dateien, I/O Kanäle, Netzwerkverbindungen, Locks, Prozessor, Bildschirm, Drucker
- Kritischer Abschnitt (engl. critical section): Programmcode von dem auf gemeinsam genutzte Ressourcen zugegriffen wird
- Wechselseitiger Ausschluss (engl. mutual exclusion): Verfahren, das anderen Prozessen (oder Threads) den Zutritt in kritischen Abschnitt verwehrt, solange ein Prozess (oder Thread) sich in solch einem befindet.

Deadlock

```
public class Deadlock extends Thread {
    Object r1=new Object();  Object r2=new Object();
    public void run() {
      synchronized (r1) {
        try { sleep(300); } catch (InterruptedException e) {}
        synchronized (r2) {}
      }
    }
    public static void main(String[] args) {
      Deadlock dl = new Deadlock();  dl.start();
      synchronized (dl.r2) {
        try { sleep(300); } catch (InterruptedException e) {}
        synchronized (dl.r1) {}
      }
      System.out.println("this smiley will not be shown: :-(")
    }
}
```

Deadlock

```
public class Deadlock extends Thread {
    Object r1=new Object();  Object r2=new Object();
    public void run() {
      synchronized (r1) {
        try { sleep(300); } catch (InterruptedException e) {}
        synchronized (r2) {}
      }
    }
    public static void main(String[] args) {
      Deadlock dl = new Deadlock();  dl.start();
      synchronized (dl.r2) {
        try { sleep(300); } catch (InterruptedException e) {}
        synchronized (dl.r1) {}
      }
      System.out.println("this smiley will not be shown: :-(")
    }
}
```

→ Lösung: Locken in gleicher Reihenfolge (geht aber nicht immer)!

Deadlock – 2

- Deadlock: Eine Situation in der eine Gruppe von Prozessen (Threads) für immer blockiert ist, weil jeder der Prozesse auf Ressourcen wartet, die von einem anderem Prozess in der Gruppe gehalten werden.
- Achtung: Deadlocks auch ohne Locks möglich, z.B.

t1	t2
`t2.join();`	
	`t1.join();`

Deadlock – 3

Notwendige Bedingungen, damit ein Deadlock entsteht (Coffman)

- Circular wait: Zwei oder mehr Prozesse bilden eine geschlossene Kette von Abhängigkeiten insoferne, dass ein Prozess auf die Ressource des nächsten Prozesses wartet.
- Hold and wait: Processe fordern neue Ressourcen an, obwohl sie den Zugriff auf andere Ressourcen behalten.
- Mutual exclusion: Der Zugriff auf die Ressourcen ist exklusiv
- No preemption: Ressourcen können Prozessen nicht entzogen werden.

Verhinderung eines Deadlocks

… indem eine der Bedingungen nicht erfüllt ist!

- Circular wait: Ressourcen werden in gleicher Reihenfolge angeordnet und so vergeben (siehe oben).
- Hold and wait: Alle Ressourcen werden auf einmal zugeteilt (wenn frei) oder Ressourcen werden zugeteilt.
- Mutual exclusion: exklusiven Zugriff z.B. durch Spooling auflösen (z.B. Drucker)
- No preemption: Ressource wird Prozess entzogen und anderem Prozess zugeteilt.

Producer/Consumer Problem

- Situation
  - `Boss` befüllt `WorkQueue` mit `WorkPacket`-Objekten (task)
  - mehrere `Worker`-Instanzen holen sich jeweils ein `WorkPacket` aus der `WorkQueue`
  - Synchronisation notwendig und einfach mittels `synchronized` sicherzustellen, aber

Producer/Consumer Problem

- Situation
  - `Boss` befüllt `WorkQueue` mit `WorkPacket`-Objekten (task)
  - mehrere `Worker`-Instanzen holen sich jeweils ein `WorkPacket` aus der `WorkQueue`
  - Synchronisation notwendig und einfach mittels `synchronized` sicherzustellen, aber
  - was ist wenn Queue voll und `Boss` kein weiteres `WorkPacket` in die Queue stellen kann?

Producer/Consumer Problem

- Situation
  - `Boss` befüllt `WorkQueue` mit `WorkPacket`-Objekten (task)
  - mehrere `Worker`-Instanzen holen sich jeweils ein `WorkPacket` aus der `WorkQueue`
  - Synchronisation notwendig und einfach mittels `synchronized` sicherzustellen, aber
  - was ist wenn Queue voll und `Boss` kein weiteres `WorkPacket` in die Queue stellen kann?
  - was ist wenn Queue leer und `Worker` sich kein neues `WorkPacket` aus der Queue holen kann?

Producer/Consumer Problem

- Situation
  - `Boss` befüllt `WorkQueue` mit `WorkPacket`-Objekten (task)
  - mehrere `Worker`-Instanzen holen sich jeweils ein `WorkPacket` aus der `WorkQueue`
  - Synchronisation notwendig und einfach mittels `synchronized` sicherzustellen, aber
  - was ist wenn Queue voll und `Boss` kein weiteres `WorkPacket` in die Queue stellen kann?
  - was ist wenn Queue leer und `Worker` sich kein neues `WorkPacket` aus der Queue holen kann?
  - polling?

Producer/Consumer Problem

- Situation
  - `Boss` befüllt `WorkQueue` mit `WorkPacket`-Objekten (task)
  - mehrere `Worker`-Instanzen holen sich jeweils ein `WorkPacket` aus der `WorkQueue`
  - Synchronisation notwendig und einfach mittels `synchronized` sicherzustellen, aber
  - was ist wenn Queue voll und `Boss` kein weiteres `WorkPacket` in die Queue stellen kann?
  - was ist wenn Queue leer und `Worker` sich kein neues `WorkPacket` aus der Queue holen kann?
  - polling?nein!
  - warten! $\rightarrow$ Monitorkonzept
- $\rightarrow$ Zustandsabhängige Steuerung (engl. condition synchronization)

Monitorkonzept

- Monitor … (klassischerweise) Sammlung von Prozeduren und Datenstrukturen, die als Einheit gruppiert sind.
- Prozesse können Prozeduren eines Monitor aufrufen,
  - aber nicht auf die internen Datenstrukturen dieses Monitors zugreifen,
  - aber es können nicht zwei Prozesse gleichzeitig in einem Monitor aktiv sein!
- Bedingungsvariable (engl. condition variables) mit zwei Operationen WAIT und NOTIFY (auch SIGNAL genannt)
- Prozedur kann nicht fortgesetzt werden $\rightarrow$ WAIT-Aufruf auf eine Bedingungsvariable $\rightarrow$ Prozess wird blockiert, geht in 'sleep' und anderer Prozess kann Monitor betreten
- Prozedur kann NOTIFY auf Bedingungsvariable aufrufen $\rightarrow$ anderer Prozess wird aufgeweckt und kann fortgesetzt werden wenn kein Prozess in Monitor!

Monitore in Java

- jedes Objekt
  - kann als Monitor agieren
  - hat einen Lock ($\rightarrow$ `synchronized`)
  - hat ein wait-set
- Zwei äquivalente Möglichkeiten von `synchronized`
  - `synchronized boolean withdraw(int amount) {...}`
  - `boolean withdraw(int amount) { synchronized (this) {...} }`
- Methoden `wait()` und `notifyAll` (in `java.lang.Object`)

Monitore in Java – 2

- `wait` muss innerhalb eines `synchronized` stehen
  - aktueller Thread wird blockiert wird und in das wait set des Objektes eingetragen
  - der Lock für das Objekt wird freigegeben (damit andere Threads in den Monitor eintreten können)
  - es wird `InterruptedException` geworfen, wenn aktueller Thread unterbrochen wird (`interrupt()` aus `java.lang.Thread`)
  - es gibt auch `wait(long millisec)`
- `notifyAll` muss innerhalb eines `synchronized` stehen
  - alle Threads aus wait set werden aufgeweckt (d.h. in runnable Zustand)
  - jeder Thread stellt sich an, um den Lock zu bekommen
  - es gibt auch `notify()` die (einen beliebigen) Thread aus dem wait set aufweckt.

WorkQueue

```
class WorkQueue {
    LinkedList queue = new LinkedList ();
    public synchronized void put(Object o) {
        // was ist, wenn die Queue voll ist?
        queue.addLast(o);
        // wie werden die wartenden Worker
        // benachrichtigt?
    }

    public synchronized Object take() {
        // was ist, wenn die Queue leer ist?
        return queue.removeFirst();
        // wie wird ein eventuell wartener Boss
        // benachrichtigt?
    }
}
```

WorkQueue – 2

```
class WorkQueue {
    LinkedList queue = new LinkedList ();
    public synchronized void put(Object o) {
        // was ist, wenn die Queue voll ist?
        queue.addLast(o);
        notifyAll();
    }

    public synchronized Object take() {
        while (queue.size() == 0) {  // while?
            wait();
        }
        return queue.removeFirst();
        // wie wird ein eventuell wartener Boss
        // benachrichtigt?
    }
}
```

Vor- und Nachteile

- Vorteile
  - in Sprache eingebaut
  - Lock wird immer freigegeben (auch wenn eine Exception auftritt)
- Nachteile
  - Es kann nicht festgestellt werden, ob ein Lock bereits vergeben ist.
  - Wenn ein Lock vergeben ist, dann blockiert jeder Versuch. Es gibt kein Time-out! Kein cancel!
  - Keine Differenzierung in lesende und schreibende Zugriffe. Keine
  - Zugriffskontrolle: jede Methode kann synchronized() für jedes Objekt ausführen. Es geht nicht, dass eine Methode a einen Lock eines Objektes erwirbt und eine Methode b diesen Lock wieder freigibt.