Threads und Synchronisierung (1)

Dämon-Threads

```
import java.io.PrintWriter;
public class Counter extends Thread
{
static PrintWriter out=new PrintWriter(System.out,true);

public void run()
{
    for(int i=0;i<4;i++)
    {
        out.println(getName()+": "+i);
        try{
            sleep(1000);
        }catch(InterruptedException e){
            out.println(e);
        }
    }
}

static public void main(String args[])
{
    out.println("Programmstart\n\n");
    Thread t1=new Counter();
    Thread t2=new Counter();
    t1.setDaemon(true);
    t2.setDaemon(true);
```

```
    t1.start();
    t2.start();
    try{
        sleep(2000);
    }catch(InterruptedException e){
        out.println(e);
    }
    out.println("\n\nProgrammende");
}

}
```

- Wie sieht die Ausgabe des Programms aus?
- Wie, wenn man die Zeilen mit setDaemon auskommentiert?
- Und wie, wenn man nur EINE Zeile auskommentiert?

Original-Testausgabe:

```
Programmstart
Thread-2: 0
Thread-3: 0
Thread-2: 1
Thread-3: 1
Programmende
Application terminated
```

Warum dies, wo wir doch in den Threads von 0 bis 3 zählen????

Testausgabe, beide setDaemons auskommentiert:

```
Programmstart
Thread-2: 0
Thread-3: 0
Thread-2: 1
Thread-3: 1
Programmende
Thread-2: 2
Thread-3: 2
Thread-2: 3
Thread-3: 3
Application terminated
```

Programmausgabe nach Ende des Programms????

Testausgabe, erstes setDaemon auskommentiert:

```
Programmstart
Thread-2: 0
Thread-3: 0
Thread-2: 1
Thread-3: 1
Programmende
Thread-2: 2
Thread-3: 2
Thread-2: 3
Thread-3: 3
Application terminated
```

Warum wird Thread-3: 3 noch ausgegeben, obwohl mit Thread-2 der letzte nicht-Dämon Thread beendet ist?

Synchronisation auf Thread-Ende:

Wie voriges Programm, nur Hauptprogramm leicht geändert

```
static public void main(String args[])
……
    t2.start();
    try{
        sleep(2000);
        t1.join();
        t2.join();
    }catch(InterruptedException e){
        out.println(e);
    }
……
```

Testausgabe:

```
Programmstart
Thread-2: 0
Thread-3: 0
Thread-2: 1
Thread-3: 1
Thread-2: 2
Thread-3: 2
Thread-2: 3
Thread-3: 3
Programmende
Application terminated
```

Endlich das gewünschte Ergebnis!

Deadlocks:

```
public class Deadlock extends Thread
{
static PrintWriter out=new PrintWriter(System.out,true);
    static int a=0;
    static int b=0;

public Deadlock(String name) {super(name);}

public void run()
{
    if(a==0)
    {
        a=1;
        for(long i=0;i<10000;i++) ;
        while(b!=0) ;
        // Hierher kommt man nie!
        a=0;
    }
    if(b==0)
    {
        b=1;
        for(long i=0;i<10000;i++) ;
        while(a!=0) ;
        // Hierher kommt man nie!
        b=0;
    }

}
```

```
static public void main(String args[])
{
    out.println("Programmstart\n\n");
    Deadlock t1=new Deadlock("Thread 1");
    Deadlock t2=new Deadlock("Thread 2");
    t1.start();
    t2.start();
    try{
        t1.join();
        t2.join();
    }catch(InterruptedException e) {
        out.println(e);
    }
    out.println("\n\nProgrammende");
}

}
```

Testoutput:

Leider unmöglich!

Schleife (Werte sind auf verschiedenen Computern anders):

- 1.000 -> terminiert (Warum eigentlich?)
- 10.000 -> Deadlock!

Synchronized:

Eine Klasse, die (vermeintlich) thread-safe ist:

```
public class Konto
{
    private int Kontostand=0;

public Konto (int stand)
{
    Kontostand=stand
}

public synchronized void ueberweise(int betrag, Konto
ziel)
{
    synchronized(ziel)
    {
        this.abheben(betrag);
        Thread.yield(); // Umschalten erzwingen!
        ziel.einzahlen(betrag);
    }
}
public synchronized void abheben(int betrag)
{   // Unbegrenzter Überziehungsrahmen
    Kontostand-=betrag;
}
public synchronized void einzahlen(int betrag)
{
    Kontostand+=betrag;
}
```

```
public int Saldo()
{
    return Kontostand;
}

}
```

- Warum kommt bei `ueberweise` "synchronized(ziel)" vor?
- Wo liegt das Problem?
- Wirkt das Problem sich auf das Ergebnis aus?

Testprogramm:

```
import java.io.PrintWriter;

public class Ueberweisungstest extends Thread
{
static PrintWriter out=new PrintWriter(System.out,true);
    public static Konto a=new Konto(100);
    public static Konto b=new Konto(100);

public void run()
{
    out.println("Vor Ueberw.: a hat "+a.Saldo()+" ECU");
    out.println("Vor Ueberw.: b hat "+b.Saldo()+" ECU");
    a.ueberweise(50,b);
    out.println("Nach Ueberw.: a hat "+a.Saldo()+" ECU");
    out.println("Nach Ueberw.: b hat "+b.Saldo()+" ECU");
}
```

```
static public void main(String args[])
{
    out.println("Programmstart\n\n");
    Thread t=new Ueberweisungstest();
    t.start();
    Thread.yield();
    out.println("Abfrage: a hat "+a.Saldo()+" ECU");
    out.println("Abfrage: b hat "+b.Saldo()+" ECU");
    try{
        t.join();
    }catch(InterruptedException e) {
        out.println(e);
    }
    out.println("\n\nProgrammende");
}

}
```

Testausgabe:

```
Programmstart

Vor Ueberw.: a hat 100 ECU
Vor Ueberw.: b hat 100 ECU
Abfrage: a hat 50 ECU // Wieso besitzt die Bank hier
Abfrage: b hat 100 ECU // nur insgesamt 150 ECU ?????
Nach Ueberw.: a hat 50 ECU
Nach Ueberw.: b hat 150 ECU

Programmende
Application terminated
```

Mutual Exclusion:

```
public class Mutex
{
    private Thread owner=null;
    private int wait_count=0;

    public synchronized boolean lock(int millis)
           throws InterruptedException
    {
        if(owner==Thread.currentThread())
            return true;
        while(owner!=null)
        {
            try
            {
                wait_count++;
                wait(millis);
            }
            finally
            {
                wait_count--;
            }
            if(millis!=0 && owner !=null)
                return false;
        }
        owner=Thread.currentThread();
        return true;
    }
```

```
    public synchronized boolean lock()
           throws InterruptedException
    {
        return lock(0);
    }

    public synchronized void unlock()
    {
        if(owner!=Thread.currentThread())
            throw new RuntimeException("Cannot unlock:
Thread is not Mutex owner");
        owner=null;
        if(wait_count>0)
            notify();
    }
}
```

Testprogramm:

```
import java.io.PrintWriter;

public class MutexTest extends Thread
{
static PrintWriter out=new PrintWriter(System.out,true);
    private static Mutex mtx=new Mutex();
    private String message;

public MutexTest (String name,String msg)
{
    super(name); message=msg;
}
```

```
public synchronized void run()
{
    try{
        mtx.lock();
    }catch(InterruptedException e) {
        out.println(e);
    }
    out.println("Achtung ("+getName()+"): ");
    Thread.yield();
    out.println(message+"!");
    mtx.unlock();
}

static public void main(String args[])
{
    out.println("Programmstart\n\n");
    MutexTest t1=new MutexTest("Thread 1","Message 1");
    MutexTest t2=new MutexTest("Thread 2","Message 2");
    MutexTest t3=new MutexTest("Thread 3","Message 3");
    t1.start(); t2.start(); t3.start();
    try{
        t1.join(); t2.join(); t3.join();
    }catch(InterruptedException e) {
        out.println(e);
    }
    out.println("\n\nProgrammende");
}

}
```

Testausgabe (mit MutEx):

```
Programmstart

Achtung (Thread 1):
Message 1!
Achtung (Thread 2):
Message 2!
Achtung (Thread 3):
Message 3!

Programmende
Application terminated
```

Testausgabe (ohne MutEx):

```
Programmstart

Achtung (Thread 1):
Achtung (Thread 2):
Achtung (Thread 3):
Message 1!
Message 2!
Message 3!

Programmende
Application terminated
```

Warum eine While-Schleife in lock(int)?

In einem besonderen Fall kann es passieren, daß zwei Threads gleichzeitig das wait verlassen:

1. Der MutEx ist belegt (Thread-1) und wird gerade vom Besitzer freigegeben. Nach Betreten der synchronized-Methode unlock, läuft bei einem anderen Thread (Thread-2) die Wartezeit ab und er wird in den Zustand "Runnable" versetzt (aber nicht gestartet, da der Monitor noch von unlock belegt wird!).
2. In unlock wird mit notify ein weiterer Thread (Thread-3) aufgeweckt und in den Zustand "Runnable" versetzt. Dieser Thread sollte jedoch unbestimmt warten (es wurde wait(0) aufgerufen). Auch dieser kann (noch) nicht weiterlaufen.
3. Sobald unlock verlassen wird, kann Thread-2 (auf Wartezeit gestartet) das wait verlassen und erhält den MutEx zugesprochen.
4. Sobald Thread-2 lock verläßt, kann Thread-3 aus dem wait herauskommen. Dieser Thread sollte jedoch unbestimmt warten und der MutEx ist zur Zeit nicht frei! Daher muß dieser Thread zurück zum wait geschickt werden. Dies kann beliebig oft hintereinander auftreten.

⇒ WHILE-Schleife nötig!

Race-Conditions:

```
import java.io.PrintWriter;
public class Race extends Thread
{
static PrintWriter out=new PrintWriter(System.out,true);
static Integer number=new Integer(0);

public Race (String name) {super(name);}

public void run()
{
    int n=number.intValue();
    for(long i=0;i<1000000;i++)   ; // Do something
    number=new Integer(n+1);
}

static public void main(String args[])
{
    out.println("Programmstart\n\n");
    Race t1=new Race("Thread 1");
    Race t2=new Race("Thread 2");
    t1.start(); t2.start();
    try{
        t1.join(); t2.join();
    }catch(InterruptedException e) {out.println(e);}
    out.println(number); // Ergebnis ausgeben
    out.println("\n\nProgramende");
}
}
```

Testausgabe:

```
Programmstart
1
Programmende
```

Wir haben mit 0 initialisiert und **zwei** Inkrementier-Threads gestartet, wieso kommt dann nur 1 heraus?

Einfache Abhilfe (nur hier!):

```
public void run()
{
    synchronized(number)
    {
        int n=number.intValue();
        for(long i=0;i<1000000;i++)
        ; // Do something
        number=new Integer(n+1);
    }
}
```

Testausgabe:

```
Programmstart
2
Programmende
```

Warum bringt eine Änderung auf public **synchronized** void run() hingegen keine Änderung?

Wissenswertes kurz zusammengefaßt

- Synchronized bedeutet zusätzlichen Aufwand, daher wird das Programm langsamer. Nur bei wirklichem Bedarf verwenden!
- Auch Methoden, die **static** definiert wurden, können das Schlüsselwort **synchronized** erhalten. Es wird dann der Monitor des Klassenobjekts verwendet.
- Synchonized Methoden können ohne Probleme rekursiv beliebig oft aufgerufen werden.
- Welcher wartende Thread bei einem notify fortgesetzt wird, ist **nicht** definiert.
- Monitors und Exceptions vertragen sich problemlos (Dies ist jedoch ein sehr schwieriger Punkt in der Implementierung, daher können Bugs nicht ausgeschlossen werden!).
- Wait und notify zur **Kommunikation** zwischen verschiedenen Threads (Bsp: A macht wait, um auf ein Ergebnis von B zu warten, dieser macht ein notify sobald er fertig ist): **EXTREME VORSICHT**!!! Es passiert sehr leicht, daß ein Thread ein notify macht, obwohl der andere Thread das entsprechend geplante wait noch nicht erreicht hat (Hier helfen auch genau ausgetüftelte Prioritäten nichts!) -> Deadlock Gefahr!

Time-Slicing

```
import java.io.PrintWriter;

public class TimeSlice extends Thread
{
static PrintWriter out=new PrintWriter(System.out,true);

public TimeSlice(String name)
{ super(name); }

public void run()
{
    out.println(getName()+" gestartet.");
    for(int i=1;i<5;i++)
    {
        long t=System.currentTimeMillis()+1000;
        while(System.currentTimeMillis()<t)
// Achtung: Hier kein sleep sondern dauernde Berechnung!
        ;
        out.println(getName()+": nach "+i+" Sekunden");
    }
    out.println(getName()+" beendet.");
}

static public void main(String args[])
{
    System.out.println("Programmstart\n\n");
    Thread[] t=new Thread[4];
```

```
    for(int i=0;i<4;i++)
    {
        t[i]=new TimeSlice("TimeSlice-Thread "+i);
//        t[i].setPriority(Thread.MIN_PRIORITY+i);
//        t[i].setPriority(Thread.MIN_PRIORITY+4-i);
        t[i].start();
        try{
            sleep(100);
        }catch(InterruptedException e){
            out.println(e);
        }
    }
    try{
        sleep(5000);
    }catch(InterruptedException e){
        out.println(e);
    }
}

}
```

- Was passiert bei gleichen Prioritäten?
- Was passiert bei steigenden Prioritäten: Absolut ansteigend oder können auch kleinere Verschiebungen vorkommen?
- Was passiert bei fallenden Prioritäten?

Testausgabe (Gleiche Prioritäten):

```
Programmstart
TimeSlice-Thread 0 gestartet.
TimeSlice-Thread 1 gestartet.
TimeSlice-Thread 2 gestartet.
TimeSlice-Thread 3 gestartet.
TimeSlice-Thread 0: nach 1 Sekunden
TimeSlice-Thread 1: nach 1 Sekunden
TimeSlice-Thread 2: nach 1 Sekunden
TimeSlice-Thread 3: nach 1 Sekunden
TimeSlice-Thread 0: nach 2 Sekunden
TimeSlice-Thread 1: nach 2 Sekunden
TimeSlice-Thread 2: nach 2 Sekunden
TimeSlice-Thread 3: nach 2 Sekunden
TimeSlice-Thread 0: nach 3 Sekunden
TimeSlice-Thread 1: nach 3 Sekunden
TimeSlice-Thread 2: nach 3 Sekunden
TimeSlice-Thread 3: nach 3 Sekunden
TimeSlice-Thread 0: nach 4 Sekunden
TimeSlice-Thread 0 beendet.
TimeSlice-Thread 1: nach 4 Sekunden
TimeSlice-Thread 1 beendet.
TimeSlice-Thread 2: nach 4 Sekunden
TimeSlice-Thread 2 beendet.
TimeSlice-Thread 3: nach 4 Sekunden
TimeSlice-Thread 3 beendet.
Application terminated
```

Testausgabe (Steigende Prioritäten):

```
Programmstart
TimeSlice-Thread 0 gestartet.
TimeSlice-Thread 1 gestartet.
TimeSlice-Thread 2 gestartet.
TimeSlice-Thread 3 gestartet.
TimeSlice-Thread 2: nach 1 Sekunden
TimeSlice-Thread 3: nach 1 Sekunden
TimeSlice-Thread 2: nach 2 Sekunden
TimeSlice-Thread 3: nach 2 Sekunden
TimeSlice-Thread 0: nach 1 Sekunden
TimeSlice-Thread 1: nach 1 Sekunden
TimeSlice-Thread 3: nach 3 Sekunden
TimeSlice-Thread 2: nach 3 Sekunden
TimeSlice-Thread 3: nach 4 Sekunden
TimeSlice-Thread 3 beendet.
TimeSlice-Thread 2: nach 4 Sekunden
TimeSlice-Thread 2 beendet.
TimeSlice-Thread 0: nach 2 Sekunden
TimeSlice-Thread 1: nach 2 Sekunden
TimeSlice-Thread 0: nach 3 Sekunden
TimeSlice-Thread 1: nach 3 Sekunden
TimeSlice-Thread 0: nach 4 Sekunden
TimeSlice-Thread 0 beendet.
TimeSlice-Thread 1: nach 4 Sekunden
TimeSlice-Thread 1 beendet.
Application terminated
```

Nicht absolut ansteigend, aber Priorität ist erkennbar!

Testausgabe (Fallende Prioritäten):

```
Programmstart
TimeSlice-Thread 0 gestartet.
TimeSlice-Thread 0: nach 1 Sekunden
TimeSlice-Thread 0: nach 2 Sekunden
TimeSlice-Thread 0: nach 3 Sekunden
TimeSlice-Thread 1 gestartet.
TimeSlice-Thread 2 gestartet.
TimeSlice-Thread 0: nach 4 Sekunden
TimeSlice-Thread 0 beendet.
TimeSlice-Thread 3 gestartet.
TimeSlice-Thread 2: nach 1 Sekunden
TimeSlice-Thread 1: nach 1 Sekunden
TimeSlice-Thread 1: nach 2 Sekunden
TimeSlice-Thread 2: nach 2 Sekunden
TimeSlice-Thread 1: nach 3 Sekunden
TimeSlice-Thread 2: nach 3 Sekunden
TimeSlice-Thread 1: nach 4 Sekunden
TimeSlice-Thread 1 beendet.
TimeSlice-Thread 2: nach 4 Sekunden
TimeSlice-Thread 2 beendet.
TimeSlice-Thread 3: nach 1 Sekunden
TimeSlice-Thread 3: nach 2 Sekunden
TimeSlice-Thread 3: nach 3 Sekunden
TimeSlice-Thread 3: nach 4 Sekunden
TimeSlice-Thread 3 beendet.
Application terminated
```

Auch hier nicht absolut fallend, aber die Beachtung der Priorität ist erkennbar!