

Verteilte Systeme - Übung

Tobias Distler, Michael Gernoth

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.informatik.uni-erlangen.de

Sommersemester 2009

Was ist ein Thread?

- ▶ Aktivitätsträger mit eigenem Ausführungskontext
 - ▶ Instruktionszähler
 - ▶ Register
 - ▶ Stack
- ▶ Alle Threads laufen im gleichen Adressbereich
 - ▶ Arbeit auf lokalen Variablen
 - ▶ Kommunikation mit anderen Threads
- ▶ Vorteile
 - ▶ Ausführen paralleler Algorithmen auf einem Multiprozessorrechner
 - ▶ Durch das Warten auf langsame Geräte (z.B. Netzwerk, Benutzer) wird nicht das gesamte Programm blockiert
- ▶ Nachteile
 - ▶ Komplexe Semantik
 - ▶ Fehlersuche schwierig

Überblick

Multithreading in Java

Threads
Synchronisation
Koordinierung

Threads in Java

- ▶ Attribute von Java-Threads
 - ▶ Priorität
 - ▶ 10 Stufen:
 - von `Thread.MIN_PRIORITY` (1)
 - über `Thread.NORM_PRIORITY` (5)
 - bis `Thread.MAX_PRIORITY` (10)
 - ▶ Lassen sich auf Betriebssystemprioritäten mappen (stark systemabhängig)
 - ▶ Daemon-Eigenschaft
 - ▶ Daemon-Threads werden für Hintergrundaktivitäten genutzt
 - ▶ Sobald alle Nicht-Daemon-Threads beendet sind, ist auch das Programm beendet
- ▶ Beachte: Thread-Attribute gehen auf neu erzeugte Threads über
 - Beispiel: Daemon-Thread *A* mit Priorität 7 erzeugt Thread *B*
 - Thread *B* ist Daemon und hat ebenfalls Priorität 7

Erzeugung von Threads in Java

java.lang.Thread

Variante 1: Unterklasse von java.lang.Thread

▶ Vorgehensweise

1. Unterklasse von java.lang.Thread erstellen
2. run()-Methode überschreiben
3. Instanz der neuen Klasse erzeugen
4. An dieser Instanz die start()-Methode aufrufen

▶ Beispiel

```
class ThreadTest extends Thread {
    public void run() {
        System.out.println("Test");
    }
}
```

```
ThreadTest test = new ThreadTest();
test.start();
```

Pausieren von Threads

sleep(), yield()

▶ Ausführung für einen bestimmten Zeitraum aussetzen

▶ Mittels sleep()-Methoden

```
static void sleep(long millis);
static void sleep(long millis, int nanos);
```

- ▶ Legt den aktuellen Thread für millis Millisekunden (und nanos Nanosekunden) „schlafen“
- ▶ Achtung: Es ist nicht garantiert, dass der Thread exakt nach der angegebenen Zeit wieder aufwacht

▶ Ausführung auf unbestimmte Zeit aussetzen

▶ Mittels yield()-Methode

```
static void yield();
```

- ▶ Gibt die Ausführung zugunsten anderer Threads auf
- ▶ Keine Informationen über die Dauer der Pause

Erzeugung von Threads in Java

java.lang.Runnable

Variante 2: Implementieren von java.lang.Runnable

▶ Vorgehensweise

1. Die run()-Methode der Runnable-Schnittstelle implementieren
2. Objekt der neuen Klasse erzeugen, das Runnable implementiert
3. Instanz von Thread erzeugen, dem Konstruktor dabei das Runnable-Objekt mitgeben
4. Am neuen Thread-Objekt die start()-Methode aufrufen

▶ Beispiel

```
class RunnableTest implements Runnable {
    public void run() {
        System.out.println("Test");
    }
}
```

```
RunnableTest test = new RunnableTest();
Thread thread = new Thread(test);
thread.start();
```

Beenden von Threads

return, interrupt()

▶ Synchron

Ausführung erreicht

- ▶ ein return aus der run()-Methode
- ▶ das Ende der run()-Methode

▶ Asynchron

▶ Mittels interrupt()-Methode

```
public void interrupt();
```

- ▶ Wird (normalerweise) von außen aufgerufen
- ▶ Führt zu
 - ▶ einer InterruptedException, falls sich der Thread gerade in einer unterbrechbaren blockierenden Operation befindet
 - ▶ einer ClosedByInterruptException, falls sich der Thread gerade in einer unterbrechbaren IO-Operation befindet
 - ▶ dem Setzen einer Interrupt-Status-Variable, die mit isInterrupted() abgefragt werden kann, sonst.

Beenden von Threads

join()

Auf die Beendigung von anderen Threads warten

- ▶ Mittels join()-Methode

```
public void join() throws InterruptedException;
```

- ▶ Beispiel

```
Runnable worker = new MyWorker();
Thread workerThread = new Thread(worker);
workerThread.start();

[...]

try {
    workerThread.join();
    worker.result();
} catch (InterruptedException ie) {
    // Unterbrechungsbehandlung fuer join()
}
```

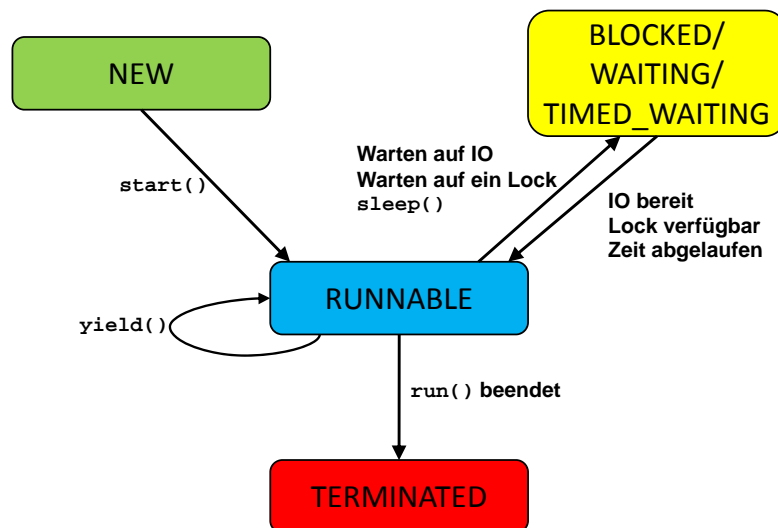
Veraltete Methoden

- ▶ Als „deprecated“ markierte Thread-Methoden
 - ▶ stop(): Thread-Ausführung stoppen
 - ▶ destroy(): Thread löschen (ohne Aufräumen)
 - ▶ suspend(): Thread-Ausführung anhalten
 - ▶ resume(): Thread-Ausführung fortsetzen
 - ▶ ...
- ▶ Gründe
 - ▶ stop() gibt alle Locks frei, die der Thread gerade hält
→ kann zu Inkonsistenzen führen
 - ▶ destroy() und suspend() geben keine Locks frei
- ▶ Weitere Informationen

“Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?”

<http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html>

Thread-Zustände in Java



Multithreading in Java

Threads
Synchronisation
Koordination

Synchronisation

State-of-the-art

Ü: „Warum ist dein Programm nicht synchronisiert?“

S: „Die Synchronisation mach' ich später!“

[später...]

S: „Kann ich die Synchronisation nicht einfach weglassen?“

Ü: „Nein.“

[später...]

S: „Ich hab' die Synchronisation weggelassen. Es geht trotzdem!“

Ü: „Zufall.“

[Bei der Abgabe...]

S: „Gerade geht's nicht. Aber vorhin hat's funktioniert!“

Ü: „Zufall.“

→ Was ist zu tun, damit es auch bei der Abgabe funktioniert?

Synchronisationsbedarf: Beispiel

```
public class Adder implements Runnable {
    public int a = 0;

    public void run() {
        for(int i = 0; i < 1000000; i++) {
            a = a + 1;
        }
    }

    public static void main(String[] args) throws Exception {
        Adder value = new Adder();
        Thread t1 = new Thread(value);
        Thread t2 = new Thread(value);

        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("Expected a = 2000000, " +
            "but a = " + value.a);
    }
}
```

Probleme mit Multithreading: Beispiel

▶ Ergebnis einiger Durchläufe: 1732744, 1378075, 1506836

▶ Was passiert, wenn `a = a + 1` ausgeführt wird?

```
LOAD a into Register
ADD 1 to Register
STORE Register into a
```

▶ Mögliche Verzahnung, wenn zwei Threads beteiligt sind

0. `a = 0;`

1. T1-load: `a = 0, Reg1 = 0`

2. T2-load: `a = 0, Reg2 = 0`

3. T1-add: `a = 0, Reg1 = 1`

4. T1-store: `a = 1, Reg1 = 1`

5. T2-add: `a = 1, Reg2 = 1`

6. T2-store: `a = 1, Reg2 = 1`

→ Die drei Operationen müssen **atomar** ausgeführt werden!

Synchronisation in Java

▶ Grundprinzip

- ▶ Vor Betreten eines kritischen Abschnitts muss ein Thread ein Sperrobjekt anfordern
- ▶ Beim Verlassen des kritischen Abschnitts wird das Sperrobjekt wieder freigegeben
- ▶ Ein Sperrobjekt wird zu jedem Zeitpunkt von nur maximal einem Thread gehalten

▶ Beachte

- ▶ In Java kann jedes Objekt als Sperrobjekt dienen
- ▶ Ein Thread kann dasselbe Sperrobjekt mehrfach halten (rekursive Sperre)

Das Schlüsselwort `synchronized`

Übersicht

- ▶ Schutz von kritischen Abschnitten per `synchronized`-Block

```
public void foo() {
    [...] // unkritische Operationen
    synchronized(<Sperrobjekt>) {
        [...] // zu schuetzender Code (krit. Abschnitt)
    }
    [...] // unkritische Operationen
}
```

- ▶ Ausweitung eines `synchronized`-Blocks auf die komplette Methode

```
synchronized public void bar() {
    [...] // zu schuetzender Code (kritischer Abschnitt)
}
```

- ▶ Verbesserung für Beispiel

```
synchronized(this) { a = a + 1; }
```

Das Schlüsselwort `synchronized`

Nachteile

- ▶ Anforderung (`lock()`) und Freigabe (`unlock()`) des Sperrobjekts sind
 - ▶ nur im Java-Byte-Code sichtbar
 - ▶ nicht trennbar (→ Vorteil: kein `lock()` ohne `unlock()`)
- ▶ Keine Timeouts beim Warten auf ein Sperrobjekt möglich
- ▶ Keine alternativen Semantiken (z.B. zur Implementierung von Fairness) definierbar

→ Lösung: Alternative Synchronisationsvarianten (siehe später)

Wann muss synchronisiert werden?

- ▶ Atomare Aufrufe erforderlich

1. Der Aufruf einer (komplexen) Methode muss atomar erfolgen

- ▶ Eine Methode enthält mehrere Operationen, die auf einem konsistenten Zustand arbeiten müssen

- ▶ Beispiele:

- „a = a + 1“

- Listen-Operationen (`add()`, `remove()`,...)

2. Zusammenhängende Methodenaufrufe müssen atomar erfolgen

- ▶ Methodenfolge muss auf einem konsistenten Zustand arbeiten

- ▶ Beispiel:

```
List list = new LinkedList();
[...]
int lastObjectIndex = list.size() - 1;
Object lastObject = list.get(lastObjectIndex);
```

- ▶ Beachte: Code, der zu jedem Zeitpunkt nur von einem einzigen Thread ausgeführt wird (*single-threaded context*), muss **nicht** synchronisiert werden!

Synchronisierte Datenstrukturen

java.util.Collections

Die Klasse `java.util.Collections`

- ▶ Idee

- ▶ Statische Wrapper-Methoden für `java.util.Collection`-Objekte
- ▶ Synchronisation kompletter Datenstrukturen

- ▶ Methoden

```
static <T> List<T> synchronizedList(List<T> list);
static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
static <T> Set<T> synchronizedSet(Set<T> s);
[...]
```

- ▶ Beispiel

```
List<String> list = new LinkedList<String>();
List<String> syncList = Collections.synchronizedList(list);
```

- ▶ Beachte

- ▶ Synchronisiert **alle** Zugriffe auf eine Datenstruktur
- ▶ Löst Fall 1, jedoch nicht Fall 2 (siehe vorherige Folie)

Multithreading in Java

Threads
Synchronisation
Koordinierung

Koordinierungsbedarf: Beispiel

Das „Philosophen-Problem“

- ▶ Erkenntnisse
 - ▶ Das Leben eines Philosophen beschränkt sich auf 2 Tätigkeiten: Denken und Essen (abwechselnd)
 - ▶ Zum Essen benötigt man Messer **und** Gabel
 - ▶ Experiment
 - ▶ 4 Philosophen werden an einem runden Tisch platziert
 - ▶ Zwischen 2 Philosophen liegt jeweils ein Messer oder eine Gabel
 - ▶ Messer und Gabel können nicht gleichzeitig genommen werden
 - ▶ Da nicht ausreichend Besteck für alle vorhanden ist, legt jeder Philosoph sein Besteck nach dem Essen wieder zurück
 - ▶ Problem, falls alle Philosophen zuerst das rechte und danach erst das linke Besteckteil nehmen wollen: **Deadlock**
- Koordinierung notwendig

Koordinierung

- ▶ Synchronisation alleine nicht ausreichend
 - ▶ Jeder Thread „lebt in seiner eigenen Welt“
 - ▶ Threads haben keine Möglichkeit sich abzustimmen
- ▶ Koordinierung unterstützt
 - ▶ Verwaltung von gemeinsam genutzten Betriebsmitteln
 - ▶ Rollenverteilung (z.B. Producer/Consumer)
 - ▶ Gemeinsame Behandlung von Problemsituationen
 - ▶ ...

Koordinierung in Java

Übersicht

- ▶ Grundprinzip
 - ▶ Ein Thread wartet darauf, dass eine Bedingung wahr wird oder ein Ereignis eintritt
 - ▶ Threads werden mittels Synchronisationsvariablen benachrichtigt
- ▶ Beachte
 - ▶ Jedes Java-Objekt kann als Synchronisationsvariable dienen
 - ▶ Um andere Threads über eine Synchronisationsvariable zu benachrichtigen, muss sich ein Thread innerhalb eines `synchronized`-Blocks dieser Variable befinden
- ▶ Methoden
 - ▶ `wait()`: auf eine Benachrichtigung warten
 - ▶ `notify()`: Benachrichtigung an **einen** wartenden Thread senden
 - ▶ `notifyAll()`: Benachrichtigung an **alle** wartenden Thread senden

Koordinierung in Java

Beispiel

Beispiel

▶ Variablen

```
Object syncObject = new Object(); // Sync.-Variable
boolean condition = false;        // Bedingung
```

▶ Auf Erfüllung der Bedingung wartender Thread

```
synchronized(syncObject) {
    while(!condition) {
        syncObject.wait();
    }
}
```

▶ Bedingung erfüllender Thread

```
synchronized(syncObject) {
    condition = true;
    syncObject.notify();
}
```

ReentrantLock VS. synchronized

Vor- und Nachteile im Vergleich

▶ ReentrantLock

- ▶ Mehr Features (Timeouts, Unterbrechbarkeit,...)
- ▶ Performanter
- ▶ Nicht an Code-Blöcke gebunden
- ▶ Schwieriger zu Debuggen

▶ synchronized

- ▶ JVM kann beim Debuggen helfen
- ▶ Einfacher zu benutzen
- ▶ Keine vergessenen unlock()s

Explizite Locks

java.util.concurrent.locks.Lock

▶ Allgemeine Schnittstelle java.util.concurrent.locks.Lock

▶ Lock anfordern

```
void lock();
void lockInterruptibly() throws InterruptedException;
boolean tryLock();
boolean tryLock(long time, TimeUnit unit)
    throws InterruptedException;
```

▶ Lock freigeben

```
void unlock();
```

▶ Condition-Variable für dieses Lock erzeugen

```
Condition newCondition();
```

▶ Implementierung: java.util.concurrent.locks.ReentrantLock

```
Lock lock = new ReentrantLock();
lock.lock();
[...]
lock.unlock();
```

Bedingungsvariablen

java.util.concurrent.locks.Condition

Die Schnittstelle java.util.concurrent.locks.Condition

▶ Auf Signal (= Erfüllung der Bedingung) warten

```
void await() throws InterruptedException; // vgl. wait()
void awaitUninterruptibly();
boolean await(long time, TimeUnit unit)
    throws InterruptedException;
boolean awaitUntil(Date deadline)
    throws InterruptedException;
```

▶ Signalisieren

```
void signal(); // analog zu notify()
void signalAll(); // analog zu notifyAll()
```

▶ Beachte

Ein Thread der await*() oder signal*() aufruft muss das zugehörige Lock halten (vgl. wait() und notify*() innerhalb synchronized-Block)

Semaphoren

`java.util.concurrent.Semaphore`

Die Klasse `java.util.concurrent.Semaphore`

▶ Konstruktoren

```
Semaphore(int permits);  
Semaphore(int permits, boolean fair);
```

▶ Semaphore belegen (= herunter zählen)

```
acquire([int permits]) throws InterruptedException;  
acquireUninterruptibly([int permits]);  
tryAcquire([int permits, ] [long timeout]);
```

▶ Semaphore freigeben (= herauf zählen)

```
release([int permits]);
```

▶ Beispiel

```
Semaphore s = new Semaphore(1);  
s.acquireUninterruptibly();  
[...]  
s.release();
```