Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf
Freie Universität Berlin

Synchronisationsmittel in Java

- Vorhanden
  - synchronized Modifikator für Methoden und Blöcke
  - wait(), notify(), notifyAll() auf Objekten
- Nachteile
  - Keine „höheren“ Synchronisationskonstrukture vorhanden
  - synchronized Monitore nur entlang der Blockstruktur
  - Versuch, eine Sperre zu erhalten nicht abbrechbar
  - Semantik von Sperren festgeschrieben (insb. Fairness)
  - Sperren auf beliebigen Objekten zugänglich
- Java 5
  - Bibliothek mit weiteren Synchronisationskonstrukturen etc.
  - „Concurrency Utilities“ in java.util.concurrent

Concurrency Utilities

- Klassen für atomaren Zugriff auf Variablen und Referenzen
  - java.util.concurrent.atomic
- Erweitertes Collections Paket optimierten nebenläufig nutzbaren Varianten
  - java.util.concurrent
- Erweiterte Sperren
  - java.util.concurrent.locks
- Erweiterte Synchronisationsmittel
  - java.util.concurrent
- …
- Siehe http://java.sun.com/j2se/1.5.0/docs/api/overview-summary.html

Atomare Variablen

- Zur Erinnerung
  - In Java ist ohne Synchronisation (volatile oder synchronized) nicht gesichert, dass
    - Lese- oder Schreibzugriffe auf long- oder double-Variable unteilbar sind,
    - die Änderung an einer Variablen von einem anderen Thread aus sofort sichtbar ist,
    - die Änderungen an mehreren Variablen von einem anderen Thread aus in der gleichen Reihenfolge beobachtet werden
  - volatile lässt sich zwar für die Referenz auf ein Feld deklarieren aber nicht für die Feldelemente
  - i++ ist teilbar auch wenn i als volatile deklariert ist

java.util.concurrent.atomic

- java.util.concurrent.atomic Klassen
  - atomaren Zugriff auf boolean, int, long und Referenz-Typen
  - Hilfsmethoden
  - typspezifische Hilfsmethoden
- Beispiel (http://java.sun.com/developer/technicalArticles/J2SE/concurrency)

```
import java.util.concurrent.atomic.*;
public class SequenceGenerator {
   private AtomicLong number = new AtomicLong(0);
   public long next() {
     return number.getAndIncrement();
   }
}
```


Klassen in java.util.concurrent.atomic

- Atomar zugreifbare Variablen und Felder davon
  - AtomicBoolean
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicLong
  - AtomicLongArray
  - AtomicReference<V>
  - AtomicMarkableReference<V>
  - AtomicStampedReference<V>
  - AtomicReferenceArray<E>
- Update mehrere Feldelemente
  - AtomicIntegerFieldUpdater<T>
  - AtomicLongFieldUpdater<T>
  - AtomicReferenceFieldUpdater<T,V>

java.util.concurrent.atomic.AtomicInteger

- Lesen
  - int get()
  - int intValue()
  - long longValue()
  - double doubleValue()
  - float floatValue()
- Modifizierendes Lesen
  - int getAndSet(int newValue)
  - int getAndIncrement() (<return i++>)
  - int incrementAndGet() (<return ++i>)
  - int getAndDecrement() (<return i-->)
  - int decrementAndGet() (<return --i>)
  - int getAndAdd(int delta) (<return i; i+=delta>)
  - int addAndGet(int delta) (<return (i+=delta)>)
- Schreiben
  - void set(int newValue)
- Bedingtes Schreiben (<return (i==expect ? (i=update) : i)>)
  - boolean weakCompareAndSet(int expect, int update) (Versuch)
  - boolean compareAndSet(int expect, int update) (sicher)

java.util.concurrent.atomic.AtomicIntegerArray

- Konstruktoren
  - AtomicIntegerArray(int length)
  - AtomicIntegerArray(int[] array)
- Lesen
  - int get(int i)
  - int length()
- Modifizierendes Lesen
  - int getAndSet(int i, int newValue)
  - int getAndIncrement(int i)
  - int incrementAndGet(int i)
  - int getAndDecrement(int i)
  - int decrementAndGet(int i)
  - int getAndAdd(int i, int delta)
  - int addAndGet(int i, int delta)
- Schreiben
  - void set(int i, int newValue)
- Bedingtes Schreiben (<return (f[i]==expect ? (f[i]=update) : f[i])>)
  - boolean compareAndSet(int i, int expect, int update)
  - boolean weakCompareAndSet(int i, int expect, int update)

Collections mit Nebenläufigkeit

- Collections ab Java 1.2
  - Schnittstellen List, Set, Map
  - Implementierungen HashMap, Hashtable, TreeMap, WeakHashMap, HashSet, TreeSet, Vector, ArrayList, LinkedList, etc.
  - Nebenläufigkeitsverträglich: Hashtable and Vector
  - Mit Collections.synchronizedMap, synchronizedList, synchronizedSet können die anderen sicher gemacht werden
  - Sperren der Collection normalerweise notwendig während Iteration (ConcurrentModificationException)
  - Schlechte Performance bei nebenläufigem Zugriff
- java.util.concurrent package
  - Führt neue nebenläufigkeitsverträgliche Implementierungen (ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet) ein. Diese sind auf Performance ausgelegt
  - Führt neue Schnittstellen ein:
    - BlockingQueue: Blockierende Entnahme

Collections aus java.util.concurrent

- Collections aus java.util benutzen *fail-fast* Iteratoren: Wenn sich Collection während der Iteration verändert, wird ConcurrentModificationException geworfen und die Iteration abgebrochen
  - Problematisch beispielsweise wenn ein Vector eine Menge von Observers verwaltet: Eine laufende Benachrichtigung über eine Änderung würde abgebrochen wenn sich ein weiterer Observer nebenläufig registriert
- Erwünscht: Iteratoren *versuchen* nur, *eine* konsistente Sicht auf die Collection zu liefern
- Collections aus java.util.concurrent benutzen *weakly consistent* Iteratoren:
  - Elemente die nach dem Start der Iteration gelöscht wurden, werden nicht durch next() ausgeliefert
  - Elemente die nach dem Start der Iteration hinzugefügt wurden, werden oder werden nicht durch next() ausgeliefert
  - Kein Element wird doppelt ausgeliefert

BlockingQueue

- Schnittstelle BlockingQueue (Unterschnittstelle von Collection<E>, Iterable<E>, Queue<E>)
  - Entnahme blockiert bis Element vorhanden
  - Ablegen blockiert bis Platz frei
- Implementierungen:
  - ArrayBlockingQueue
    Implementierung basierend auf einem (begrenzten) Feld
  - LinkedBlockingQueue
    Implementierung basierend auf (unbegrenzt vielen) verketteten Elementen
  - DelayQueue
    Elemente müssen eine bestimmte Zeit in der Schlange verbringen und können erst danach entnommen werden.
  - PriorityBlockingQueue
    Implementierung, die nach einer Rangordnung Elemente ausliefert (siehe PriorityQueue)
  - SynchronousQueue
    Implementierung mit Puffergröße=0
    - put() wartet auf take()
    - take() wartet auf put()

Erweiterte Sperren

- Sperren mit synchronized sind
  - an Blockstruktur des Programms gebunden
  - in ihrem Verhalten (z.B. Fairness) nicht veränderbar
  - nicht sehr ausdrucksmächtig
- java.util.concurrent.locks führt ein
  - Sperroperationen
  - Verschiedene Implementierungen

Lock Schnittstelle

- Vorgesehene Methoden:
  - void lock()
    Sperre anfordern (blockiert)
  - void lockInterruptibly()
    Sperre anfordern, solange Thread nicht unterbrochen wird
  - boolean tryLock()
    Sperre anfordern, wenn Sperre frei
  - boolean tryLock(long time, TimeUnit unit)
    Sperre für einen bestimmten Zeitraum anfordern
  - void unlock()
    Sperre aufgeben
- Idiom:

```
Lock l = ...;
l.lock();
try {
  // access the resource protected by this lock
} finally {
  l.unlock();
}
```


ReentrantLock

- Entspricht den implizit bei jedem Objekt vorhandenen Sperren, aber explizit als Objekt
- Konstruktoren:
  - ReentrantLock()
  - ReentrantLock(boolean fair)
    Neue Sperre mit Fairness-Einstellung
- Zusätzliche Methoden:
  - isLocked()
    Hat ein Thread die Sperre?
  - protected Thread getOwner()
    Welcher Thread hat diese Sperre
  - boolean isHeldByCurrentThread()
    Hat dieser Thread die Sperre
  - int getHoldCount()
    wie oft hat der Thread die Sperre bekommen?
  - protected Collection<Thread> getQueuedThreads()
    Welche Threads wollen diese Sperre?
  - boolean hasQueuedThread(Thread thread)
    Will jener Thread diese Sperre

ReentrantReadWriteLock

- Lese-/Schreibsperren
- Konstruktoren
  - ReentrantReadWriteLock()
  - ReentrantReadWriteLock(boolean fair)
    Neue Sperre mit Fairness-Einstellung
- Zusätzliche Methoden:
  - ReentrantReadWriteLock.ReadLock readLock()
    Lesesperre erhalten
  - ReentrantReadWriteLock.WriteLock writeLock()
    Schreibsperre erhalten
- Verhalten
  - writeLock().lock() blockiert wenn andere Lese- oder Schreibsperren gesetzt sind
  - readLock().lock() blockiert wenn andere Schreibsperren gesetzt sind

Condition Objekte

- An Objektsperren gibt es nur eine einzige Bedingung auf die mit wait() gewartet werden kann
- Condition Objekte machen diese Bedingung explizit
- Ein Sperrobjekt kann mehrere Bedingungsobjekte haben
- In der Lock Schnittstelle weiterhin:
  Condition newCondition()
  Erzeugt und bindet ein Condition-Objekt an Sperre
- Methoden
  - void await()
  - boolean await(long time, TimeUnit unit)
  - boolean awaitUntil(Date deadline)
    Auf Signal warten
  - void signal()
    Entspricht notify()
  - void signalAll()
    Entspricht notifyAll()
- Können fair sein

Beispiel

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
```


Beispiel

```
  public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      Object x = items[takeptr];
      if (++takeptr == items.length) takeptr = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```


Synchronisationsmittel

- Java 5 führt über Monitore und Signale hinaus weitere Synchronisationsmittel als Klassen ein:
  - Semaphore
  - CyclicBarrier
  - CountdownLatch
  - Exchanger
  - ...

Semaphore

- Konstruktoren
  - Semaphore(int permits)
  - Semaphore(int permits, boolean fair)
- P Operation:
  - void acquire()
    Ein Freisignal nehmen (blockierend)
  - void acquire(int permits)
    permits Freisignale nehmen (blockierend)
  - int drainPermits()
    Alle gerade freien Freisignal nehmen (nicht blockierend)
  - boolean tryAcquire()
    Ein Freisignal nehmen (nicht blockierend)
  - boolean tryAcquire(int permits)
    permits Freisignale nehmen (nicht blockierend)
  - boolean tryAcquire(long timeout, TimeUnit unit)
    Ein Freisignal nehmen (blockierend mit Timeout)
  - boolean tryAcquire(int permits, long timeout, TimeUnit unit)
    permits Freisignale nehmen (blockierend mit Timeout)
- V Operation:
  - void release()
    Ein Freisignal freigeben
  - void release(int permits)
    permits Freisignale freigeben

Beispiel

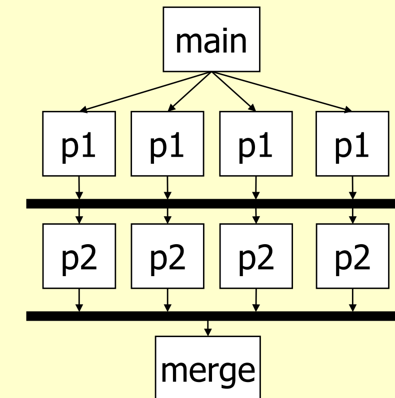
```
class Pool {
  private static final MAX_AVAILABLE = 100;
  private final Semaphore available =
   new Semaphore(MAX_AVAILABLE, true);

  public Object getItem() throws InterruptedException {
    available.acquire();
    return getNextAvailableItem();
  }

  public void putItem(Object x) {
    if (markAsUnused(x))
      available.release();
  }
...
```


CyclicBarrier

- Barrierensynchronisierung:
  Prozesse arbeiten nebenläufig bis sie sich vor der Weiterverarbeitung an einer Barriere treffen/synchronisieren. Sind alle da, kann nach der Barriere weitergearbeitet werden
- CyclicBarrier:
  Wenn eine „Herde" Prozesse passiert hat können sich weitere Prozesse sammeln



CyclicBarrier

- Konstruktoren
  - CyclicBarrier(int parties)
    Wenn sich parties Prozesse gesammelt haben Barriere aufheben
  - CyclicBarrier(int parties, Runnable barrierAction)
    mit barrierAction fortfahren nach Aufheben der Barriere
- Methoden
  - int await()
    Warten bis Barriere aufgehoben wird
  - int await(long timeout, TimeUnit unit)
    Warten mit Timeout
  - int getNumberWaiting()
    Wieviele warten
  - int getParties()
    Wieviele müssen da sein?
  - void reset()
    Zurücksetzen, neuer Zyklus

Beispiel

```
public class Summary {
   private static int matrix[][] = {
     {1}, {2, 2}, {3, 3, 3},
     {4, 4, 4, 4}, {5, 5, 5, 5, 5}
   };
   private static int results[];
   public static void main(String args[]) {
     final int rows = matrix.length;
     results = new int[rows];
     Runnable merger = new Runnable() {
       public void run() {
         int sum = 0;
         for (int i=0; i<rows; i++) {
           sum += results[i];
         }
         System.out.println("Results are: " + sum);
       }
     };
     CyclicBarrier barrier = new CyclicBarrier(rows, merger);
     for (int i=0; i<rows; i++) {
       new Summer(barrier, i).start();
     }
     System.out.println("Waiting...");
   }
}
```

Beispiel

```
private static class Summer extends Thread {
  int row;
  CyclicBarrier barrier;
  Summer(CyclicBarrier barrier, int row) {
    this.barrier = barrier;
    this.row = row;
  }
  public void run() {
    int columns = matrix[row].length;
    int sum = 0;
    for (int i=0; i<columns; i++) {
      sum += matrix[row][i];
    }
    results[row] = sum;
    System.out.println("Results for row " + row + " are : " + sum);
    // wait for others
    try {
      barrier.await();
    } catch (InterruptedException ex) { ex.printStackTrace();
    } catch (BrokenBarrierException ex) { ex.printStackTrace(); }
  }
}
```

```
Results for row 0 are : 1
Results for row 1 are : 4
Results for row 2 are : 9
Waiting...
Results for row 3 are : 16
Results for row 4 are : 25
Results are: 55
```


CountDownLatch

- CountDownLatch
  - ist ein Zähler
  - kann von Prozessen dekrementiert werden
  - man kann auf den Zählerstand 0 blockierend warten
- Konstruktor
  - CountDownLatch(int count)
- Methoden
  - void await()
    Auf Nullstand warten
  - boolean await(long timeout, TimeUnit unit)
    Auf Nullstand warten mit Timeout
  - void countDown()
    Dekrementieren
  - long getCount()
    Aktueller Zählerstand

Beispiel: Start- und Stopsignale

```
class Driver { // ...
  void main() throws InterruptedException {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(N);
    for (int i = 0; i < N; ++i) // create and start threads
      new Thread(new Worker(startSignal, doneSignal)).start();
    doSomethingElse();           // don't let run yet
    startSignal.countDown();      // let all threads proceed
    doSomethingElse();
    doneSignal.await();          // wait for all to finish
  }
}
class Worker implements Runnable {
  private final CountDownLatch startSignal;
  private final CountDownLatch doneSignal;
  Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
     this.startSignal = startSignal;
     this.doneSignal = doneSignal;
  }
  public void run() {
     try {
       startSignal.await();
       doWork();
       doneSignal.countDown();
     } catch (InterruptedException ex) {} // return;
  }
  void doWork() { ... }
}
```


Asynchrone Aufrufe

- Future ist das Resultat einer asynchronen Berechnung
- Lesezugriff darauf blockiert falls Ergebnis noch nicht vorliegt
- Schnittstelle Future<V>
  - V get()
    Wert auslesen (blockierend)
  - V get(long timeout, TimeUnit unit)
    Wert auslesen mit Timeout
  - boolean cancel(boolean mayInterruptIfRunning)
    Berechnung abbrechen
  - boolean isCancelled()
    Wurde abgebrochen?
  - boolean isDone()
    Liegt Ergebnis vor?
- Wo kommen Futures her?
  - Mittel zum Start einer asynchronen Berechnung nötig

Asynchrone Aufrufe

- FutureTask ist Berechnung, die als Thread gestartet werden kann
- Konstruktor
  - FutureTask(Runnable runnable, V result)
    Berechnung erzeugen (nicht starten!)
  - FutureTask(Callable<V> callable)
    (Callable-Schnittstelle definiert V call())
- Methoden
  - V get()
  - V get(long timeout, TimeUnit unit)
  - boolean cancel(boolean mayInterruptIfRunning)
  - boolean isCancelled()
  - boolean isDone()
  - protected void done()
    Ergebnis liegt vor
  - void run()
    Berechnung starten
  - protected void set(V v)
    Futurewert setzen
  - protected void setException(Throwable t)
    Ausnahme setzen, die die Abfrage des Futures werfen soll

 © Robert Tolksdorf, Berlin

Ende

- Stöbern Sie in den util.concurrent Paketen und Klassen
- Aus ALP IV müssten Sie wissen
  - Wie man die vorhandenen Objekte benutzt
  - Wie man sie implementiert
  - Welche Mittel der nebenläufigen Programmierung nicht in Java vorliegen

 © Robert Tolksdorf, Berlin