

Building Blocks

Proseminar Nebenläufige Programmierung
Julia Polleti, Christine Wagner

Gliederung

- 1) Synchronized collections
 - 2) Concurrent collections
 - 3) Blocking Queues und das Producer-Consumer-Pattern
 - 4) Blocking und interruptible Methoden
 - 5) Synchronizers
 - 6) Erstellung eines effizienten, skalierbaren Resultat-Cache
-

1) synchronized collections

Thread-safety

- Einkapselung
 - Synchronisierung jeder public Methode
-

synchronized collections

A) Probleme

- Verwendung von compound actions (iteration, navigation, conditional operations)
→ oft nur theoretisch thread-safe

Lösung:

→ client-side-locking

Aber: Einschränkung der Skalierbarkeit

Codebeispiel: check-then-act

```
public static Object getLast(Vector list) {  
    synchronized(list) {  
        int lastIndex = list.size()-1;  
        return list.get(lastIndex);  
    }  
}
```

```
public static void deleteLast(Vector list) {  
    synchronized(list) {  
        int lastIndex = list.size()-1;  
        list.remove(lastIndex);  
    }  
}
```

Codebeispiel: iteration

```
for (int i=0; i<vector.size(); i++)  
    doSomething(vector.get(i));
```

- kann `ArrayIndexOutOfBoundsException` werfen
- Lösung: client-side-locking?

Nicht sinnvoll: verschlechterte Nebenläufigkeit,
lange Wartezeit

synchronized collections

B) Iteratoren

- fail-fast: werfen `ConcurrentModificationException`, wenn Änderung während der Iteration festgestellt wird
- Modification-count: zählt Veränderungen ab Beginn der Iteration
Aber: nicht synchronisiert
→ Risiko: Iterator merkt Änderung nicht

Lösung?

synchronized collections

- Synchronisieren nicht günstig, weil:
 - Lange Wartezeiten
 - Deadlock
 - Verschlechterung der Skalierbarkeit
 - Auslastung leidet

 - Alternative:
 - Collection kopieren
 - Iterator durchläuft Kopie
 - Abhängig von verschiedenen Faktoren
-

synchronized collections

C) Versteckte Iteratoren

- Iteratoren oft nicht auf den ersten Blick erkennbar

```
Public class HiddenIterator {
    @GuardedBy(„this“)
    private final Set<Integer> set=new HashSet<Integer>();
    public synchronized void add(Integer i){set.add(i);}
    public synchronized void remove(Integer i){set.remove(i);}
    public void addTenThings(){
        Random r = new Random();
        for(int i=0; i<10; i++)
            add(r.nextInt());
        System.out.println(„DEBUG: added ten elements to“ + set);
    }
}
```

synchronized collections

Problem:

Versteckte Iteratoren können immer
ConcurrentModificationException werfen!

Weitere Methoden:

equals, hashCode, removeAll, containsAll, retainAll

2) concurrent collections

- synchronized
 - Thread-safe durch Synchronisieren
 - Beeinträchtigung der Nebenläufigkeit
- Ersetzung von synchronized collections durch concurrent collections verbessert die Skalierbarkeit und Nebenläufigkeit stark

synchronized	concurrent
HashMap	ConcurrentHashMap
List	CopyOnWriteArrayList
Set	ConcurrentSet

concurrent collections

A) ConcurrentHashMap

- **Problem** von *HashMap*: hält Lock für gesamte Dauer einer Operation → lange Laufzeit
- **Lösung** von *ConcurrentHashMap*: Lock striping

Lock striping

Ermöglicht paralleles Lesen
und Schreiben durch
mehrere Threads

concurrent collections

- **Iteratoren:** weakly-consistent anstatt fail-fast
→ können Veränderungen tolerieren und zurückgeben, müssen aber nicht
- **Kompromiss:** Methoden, wie size oder isEmpty geben nur Annäherung zurück, anstatt exakter Zahlen

Aber: Fokus auf Optimierung der wichtigsten Operationen (get, put, containsKey, remove)

concurrent collections

- **Allerdings:** bei synchronized collections kann die ganze Map für Einzelzugriff gesperrt werden, bei concurrent collections nicht
-

concurrent collections

B) CopyOnWriteArrayList

Thread-safety:

- Keine Synchronisation nötig, solange Objekt unveränderlich
- Erstellung einer Kopie der collection vor jeder Veränderung

Nachteil? Aufwand bei langen Listen zu groß

→ eher für Lesevorgang geeignet

3) Blocking Queues und das Producer-Consumer-Pattern

A) Producer-Consumer-Pattern

- **Producer:** Identifikation der auszuführenden Arbeit
 - **Consumer:** Ausführung der Aufgaben aus der Schlange
 - **Queue:** Speicherglied zwischen Producer und Consumer und koordiniert die Ablaufsteuerung
 - Producer und Consumer wissen nichts über Identität des anderen
 - Vorteil: Bessere Auslastungsverteilung
-

Blocking Queues und das Producer-Consumer-Pattern

B) BlockingQueue

- wenn Schlange leer ist, blockiert die Abfrage, bis ein Element vorliegt; ist sie voll, wird das Einfügen blockiert bis wieder Platz ist
 - unterstützt producer-consumer-pattern
 - consumer=producer bei aneinander gereihten Schlangen
 - begrenzt oder unbegrenzt
-

Blocking Queues und das Producer-Consumer-Pattern

C) Serial thread confinement

- Objekt ist auf einen Besitzer-Thread beschränkt
 - Nur dieser kann es verändern
 - Besitz kann an anderen Thread übergeben werden
 - Bedingung: Alter Besitzer darf dann nicht mehr darauf zugreifen
-

Blocking Queues und das Producer-Consumer-Pattern

D) Deques und work stealing

Deque: Schlange, bei der man an beiden Enden Elemente effizient einfügen und entfernen kann

Work stealing:

- jeder Consumer hat eigene Deque
 - Wenn diese leer ist, kann er Arbeit vom Ende einer fremden Deque stehlen
 - Sicherstellung, dass jeder Thread beschäftigt bleibt
 - Skalierbarer als producer-consumer-design
 - Besonders geeignet, wenn Consumer=Producer
-

4) blocking und interruptible Methoden

Gründe für blocking:

- Warten auf Fertigstellung von E/A
- Warten auf Erhalten eines Locks
- Warten auf Ergebnis eines anderen Threads
- Warten auf Aufwachen aus sleeping Zustand

→ Thread wieder im Runnable Zustand

blocking und interruptible Methoden

Blocking Methoden:

- Werfen InterruptedException
- Werden unterbrochen → stoppen das Blockieren
- Brauchen Plan um auf Unterbrechung zu reagieren:
 - a) Mitteilung über Exception

 - b) Zurücksetzen des Unterbrechungsstatus durch Auffangen der Exception und Aufrufen der interrupt Methode

Sonst: Hinweis auf Unterbrechung verloren, höherer Code kann nichts dagegen tun!

5) synchronizers

- Jedes Objekt, das Kontrollfluss von Threads koordiniert

Beispiele: Latches, Semaphore, Barriers

synchronizers

A) Latches

- Verzögert das Starten von Threads, bis Latch Endzustand erreicht
 - Agiert wie Schranke
 - Wenn Endzustand erreicht, Schranke für immer offen
 - Schrankenöffnung: wenn alle Threads bereit sind
Bsp.: Multiplayer
-

```

public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task)
        throws InterruptedException {
        final CountdownLatch startGate = new CountdownLatch(1);
        final CountdownLatch endGate = new CountdownLatch (n Threads);

        for(int i=0; i< nThreads; i++) {
            Thread t = new Thread() {
                public void run() {
                    try {
                        startGate.await();
                        try{
                            task.run();
                        } finally {
                            endGate.countDown();
                        }
                    } catch (InterruptedException ignored) {}
                }
            };
            t.start();
        }
        long start =System.nanoTime ();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end-start;
    }
}

```

synchronizers

B) FutureTask

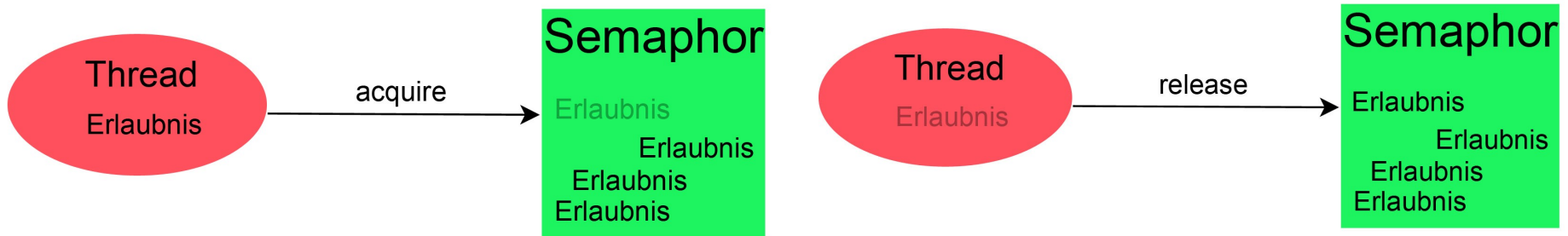
- Agiert wie Platzhalter
- Führt Operationen aus, deren Ergebnisse erst später gebraucht werden
- Übergibt Ergebnis sicher an den nächsten Thread

→ verkürzte Wartezeit

synchronizers

C) Semaphore

- Zählsemaphore: kontrollieren Anzahl gleichzeitiger Aktivitäten auf ein Objekt



synchronizers

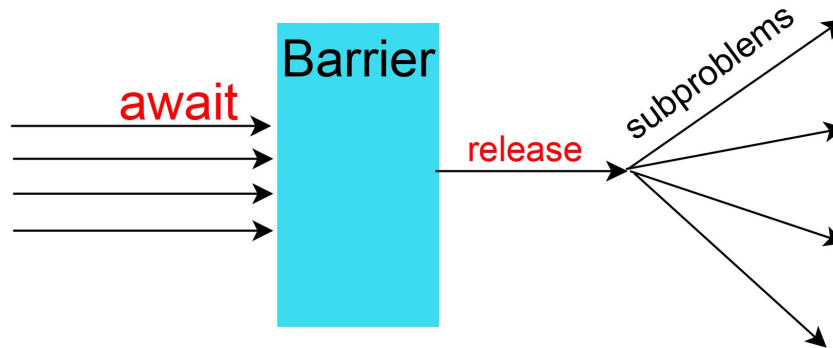
- Binärsemaphore

Vorteile:

- Nützlich bei Implementierung von Ressourcen Pools
-

synchronizers

D) Barriers



- Ähnlich wie Latch, aber: reset möglich
 - Threads müssen barrier point zur selben Zeit erreichen
 - Problem wird in viele kleine Einzelprobleme unterteilt
-

synchronizers

- **barrier action:** vermittelt errechnete Werte/Veränderungen an das Datenmodell
 - **Anwendung:** Simulation, zB Game of Life
-

6) Erstellung eines effizienten, skalierbaren Resultat-Cache

Cache:

- Zwischenspeicherung von vorhergehenden Resultaten
→ Verkürzung der Wartezeit, Erhöhung des Datenflusses
aber: höherer Speicherplatzverbrauch
-

```
public class Memoizer1<A,V> implements Computable<A,V>{
    @GuardedBy(„this“)
    private final Map<A,V> cache = new HashMap<A,V>();
    private final Computable<A,V> c;

    public Memoizer1(Computable<A,V> c) {
        this.c=c;
    }

    public synchronized V compute(A arg) throws
    InterruptedException {
        V result = cache.get(arg);
        if (result == null){
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

```
public Memoizer2<A,V> implements Computable<A,V> {
    private final Map<A,V> cache = new
        ConcurrentHashMap<A,V>();
    private final Computable<A,V> c;

    public Memoizer2(Computable<A,V> c) {this.c= c;}

    public V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result = null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

```
public class Memoizer3<A,V> implements Computable<A,V>{
    private final Map<A, Future<V>> cache =
        new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A,V> c;

    public Memoizer3(Computable<A,V> c) {this.c =c;}

    public V compute(final A arg) throws InterruptedException{
        Future<V> f= cache.get(arg);
        if(f == null){
            Callable<V> eval= new Callable<V>() {
                public V call() throws InterruptedException{
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft= new FutureTask<V>(eval);
            f= ft;
            cache.put(arg, ft);
            ft.run();
        }
        try { return f.get(); }
        catch (ExecutionException e){
            throw launderThrowable(e.getCause());
        }
    }
}
```

. . .

```
FutureTask<V> ft= new FutureTask<V>(eval) ;  
f = cache.putIfAbsent(arg, ft) ;  
if (f== null) { f= ft; ft.run();}
```

. . .