

Bakkalaureatsarbeit
Bakkalauratsseminar WS 2001/02

29. Jänner 2003

util.concurrent

Harald Röck
hroeck@cosy.sbg.ac.at

Betreuer Dr. Helge Hagenauer

Institut für Computer Wissenschaften
Universität Salzburg

Inhaltsverzeichnis

| | | |
|----------|----------------------------|-----------|
| 1 | Einführung | 3 |
| 1.1 | Sync | 3 |
| 1.2 | Channel | 3 |
| 1.3 | Executor | 4 |
| 1.4 | Barrier | 4 |
| 2 | Sync | 4 |
| 2.1 | Mutex | 5 |
| 2.1.1 | Beispiel | 5 |
| 2.2 | Latch/CountDown | 6 |
| 2.2.1 | Latch | 6 |
| 2.2.2 | CountDown | 6 |
| 2.2.3 | Beispiel | 6 |
| 2.3 | ReadWriteLock | 7 |
| 2.3.1 | Implementierungen | 8 |
| 2.3.2 | Beispiel | 8 |
| 2.4 | Semaphore | 8 |
| 2.4.1 | Unterklassen | 8 |
| 3 | Channel | 9 |
| 3.1 | Direkt Implementierungen | 9 |
| 3.1.1 | LinkedQueue | 9 |
| 3.1.2 | WaitFreeQueue | 9 |
| 3.2 | BoundedChannel | 10 |
| 3.2.1 | Implementierungen | 10 |
| 3.3 | Beispiel | 10 |
| 4 | Executor | 11 |
| 4.1 | Einfache Implementierungen | 11 |
| 4.1.1 | DirectExecutor | 11 |
| 4.1.2 | LockedExecutor | 11 |
| 4.1.3 | ThreadedExecutor | 11 |
| 4.2 | PooledExecutor | 11 |
| 4.2.1 | Beispiel | 12 |

| | | |
|----------|---|-----------|
| 5 | Barrier | 13 |
| 5.1 | CyclicBarrier | 13 |
| 5.2 | Rendezvous | 14 |
| 5.2.1 | Rendezvous.RendezvousFunction | 14 |
| 6 | Santa Claus Problem | 14 |
| 6.1 | Problemstellung | 15 |
| 6.2 | Implementierung | 15 |
| 6.2.1 | Gruppenbildung | 15 |
| 6.2.2 | Santa wecken | 15 |
| 6.2.3 | ein-/ausspannen | 16 |
| 6.3 | Code | 16 |
| 7 | Zusammenfassung | 21 |

1 Einführung

Das Java Paket `EDU.oswego.cs.dl.util.concurrent`^[2] von Doug Lea enthält einige standardisierte Klassen, die die Entwicklung von multithreaded Programmen in Java erleichtern soll. Es wurde in Verbindung zum Buch “Concurrent Programming in Java, 2nd ed.” entwickelt. In zukünftigen JDK Releases sollte dieses Paket in ähnlicher Form enthalten sein. Es läuft zur Zeit die Specification Request(JSR 166) unter der Leitung von Doug Lea.

Das Packet besteht aus verschiedenen Implementierungen der folgenden einfachen Interfaces:

- `Sync`
- `Channel`
- `Executor`
- `Barrier`
- `java.util.Collection`

In dieser Arbeit werde ich vor allem auf die ersten vier genauer eingehen. Ich werde einige Klassen, die diese Interfaces implementieren vorstellen und ihre Verwendung jeweils anhand eines oder mehrerer Beispiele erläutern.

Unten ist zu erkennen, dass die verschiedenen Methoden meistens eine `InterruptedException` auslösen. Das bedeutet, dass diese Methoden immer in einem `try/catch` Block eingeschlossen werden sollten.

1.1 Sync

```
public interface Sync{
    public void acquire() throws InterruptedException;
    public boolean attempt(long ms) throws InterruptedException;
    public void release();
}
```

`Sync` ist das Interface für Locks, Gates und Conditions. `Sync` Objekte synchronisieren das Zugreifen auf gemeinsam genutzte Ressourcen.

1.2 Channel

```
public interface Channel extends Puttable, Takable {
    public void put(Object item) throws InterruptedException;
    public boolean offer(Object item, long ms) throws InterruptedException;
    public Object take() throws InterruptedException;
    public Object poll(long ms) throws InterruptedException;
    public Object peek();
}
```

`Channel` wird für Buffers, Queues, Pipes etc verwendet. Dieses Interface wird in Klassen implementiert in die ein Thread Objekte hineingeben kann und ein anderer Thread sie wieder heraus holt.

1.3 Executor

```
public interface Executor {
    public void execute(Runnable command) throws InterruptedException;
}
```

Objekte die `Executor` implementieren stellen eine Erweiterung zu der `Thread` Klasse dar. Dadurch sollte es möglich sein die Verwaltung von mehreren Threads zu erleichtern und um einige Funktionen zu erweitern.

1.4 Barrier

```
public interface Barrier {
    public int parties();
    public boolean broken();
}
```

Barriers sind Synchronisationspunkte einer Gruppe von Threads. Dieses Interface ist sehr allgemein und seine Implementierungen, `Rendezvous` und `CyclicBarrier`, unterscheiden sich in ihrem Verhalten und Verwendung deutlich.

2 Sync

```
public interface Sync{
    public void acquire() throws InterruptedException;
    public boolean attempt(long ms) throws InterruptedException;
    public void release();
}
```

Mit `acquire` wird normalerweise eine Resource belegt. Wenn diese nicht verfügbar ist blockt die Anweisung bis ein anderer Thread ein `release` aufruft. Mit `attempt` wird versucht die Resource zu belegen, und nach `ms` Millisekunden wird der Versuch spätestens abgebrochen. `attempt` liefert `true` falls innerhalb der angegebenen Zeit die Resource belegt wurde, sonst `false`. Falls es nötig sein sollte, dass dabei eine `TimeoutException` ausgelöst werden sollte, kann man die Klasse `TimeoutSync`, eine Wrapperklasse die folgende Anweisung verwendet:

```
if (!c.attempt(timeval)) throw new TimeoutException(timeval);
```

Verwendet sollten `Sync` wie folgendes Beispiel zeigt: Die `acquire` Anweisung sollte in einem `try/catch` Block sein, und danach sollte wieder in einem neuen `try/catch/finally` Block der kritische Bereich sein. Die `release` Anweisung sollte immer im `finally` Teil dieses Blockes aufgerufen werden, damit garantiert wird, dass nach einem `acquire` auch ein `release` stattfindet.

```
Sync lock = ...
try {
    lock.acquire();
    try {
        action();
    } finally {
        lock.release();
    }
} catch (InterruptedException ie) { ... }
```

2.1 Mutex

Die `Mutex` Klasse ist eine einfache Implementierung eines mutual exclusion locks. Sie enthält nur die Methoden des `Sync` Interfaces. Bei der Instantiierung ist der Lock frei. Jedes `acquire` erhält den Lock und `release` gibt ihn frei. Wird auf einen bereits freigegebenen Lock nochmals ein `release` angewendet, hat dies keine Auswirkungen.

Es ist weiters auch nicht definiert welche wartenden Threads geweckt werden. Dies hängt von der jeweiligen JVM ab. Das bedeutet es könnte zu race conditions kommen. Sollte es jedoch nötig sein, dass irgendeine Reihenfolge einzuhalten ist, muß eine entsprechende Semaphore verwendet werden.

Der Vorteil dieses Locks im Vergleich zu den Java `synchronized` Blöcken liegt wohl vorallem an der Möglichkeit mit Timeouts zu arbeiten. Dadurch können Backoffs zur Vermeidung von Deadlocks verwendet werden:

2.1.1 Beispiel

Folgende Beispielklasse enthält eine Membervariable und eine Methode `swapVal`, die mit einem Objekt des selben Typs den Wert der Membervariable vertauscht. Dabei wird zuerst der eigene Lock gesperrt, dann wird versucht den Lock des anderen Objekts zu sperren. Falls dies gelingt werden die Werte vertauscht und abgebrochen, andernfalls wird der eigenen Lock wieder freigegeben und eine bestimmte Zeit gewartet um es dann erneut zu versuchen.

```
class UsingBackoff {
    private long val;
    private final Mutex mutex = new Mutex();

    void swapVal(UsingBackoff other) throws InterruptedException {
        if ( this == other ) return;
        for(;;) {
            mutex.acquire();
            try {
                if (other.mutex.attempt(0)) {
                    try {
                        long t = val;
                        val = other.val;
                        other.val = t;
                        return;
                    } finally {
                        other.mutex.release();
                    }
                }
            } finally {
                mutex.release();
            }
            Tread.sleep(100);
        }
    }
}
```

2.2 Latch/CountDown

2.2.1 Latch

sind Bedingungen die, wenn sie einmal gesetzt sind, immer gesetzt bleiben. Das bedeutet alle `acquire` Aufrufe blocken, bis ein `release` aufgerufen wird. Danach können `acquire` immer passieren.

2.2.2 CountDown

ist Latch sehr ähnlich. Nur das hierbei eine bestimmte Anzahl von `releases` nötig sind um die `acquire` freizugeben.

2.2.3 Beispiel

Wir haben zwei Klassen, Worker und Driver. Die Driver Klasse enthält die `main` Methode und instantiiert N Worker. Nach dem Starten der Worker kann irgendeine Arbeit verrichtet werden. Danach wird der Latch freigegeben und auf den CountDown gewartet. Die Worker warten nach dem Start bis sie das Signal, durch eine Latch erhalten. Wenn sie ihre Arbeit abgeschlossen haben setzen sie den CountDown herab. Haben alle Worker ihre Arbeit abgeschlossen, kann der Driver weitermachen.

```
class Worker implements Runnable {
    Latch startSignal;
    Countdown ready;
    Worker (Latch l, Countdown c) { startSignal = l; ready = c;}

    public void run() {
        startSignal.acquire();
// .. do some Work
        ready.release();
    }
}

class Driver{ // ..
    void main() {
        Latch ss = new Latch();
        Countdown rr = new Countdown(N);
        for (int i = 0; i < N; i++)
            new Thread( new Worker(ss, rr)).start();
//.. do something until car breaks
        ss.release();
//.. drink Coffee, and wait until workers are finished
        rr.acquire();
// ...
    }
}
```

2.3 ReadWriteLock

```
interface ReadWriteLock() {
    Sync readLock();
    Sync writeLock();
}
```

ReadWriteLock ist zwar ein eigenes Interface, es ist jedoch eine Verwendung vom Sync Interface weshalb es hier erwähnt werden soll.

ReadWriteLock verwaltet ein Paar von Locks, ein readLock und writeLock. Der readLock kann von mehreren Threads gleichzeitig gehalten werden, der writeLock kann allerdings immer nur einmal von einem Thread gehalten werden. Während eines writeLocks können auch keine readLock angefordert werden. Normalerweise sind auch immer mehrere Threads als Reader beteiligt, und meistens nur ein oder wenige Threads als Writer.

Die verschiedenen Implementierungen unterscheiden sich in der Lock policy. Die wichtigste und am meisten verwendete ist WriterPreferenceReadWriteLock. Hier wird immer ein writeLock bevorzugt. Es ist weiters darauf zu achten dass keine Reentrants möglich sind. Besitzt ein Thread bereits einen Lock, so sollte er keinen anderen anfordern. Das könnte zu Deadlocks führen oder anderen unerwünschten Auswirkungen haben.

2.3.1 Implementierungen

- `WriterPreferenceReadWriteLock`
- `FIFOReadWriteLock`
- `ReaderPreferenceReadWriteLock`
- `ReentrantWriterPreferenceReadWriteLock` – hiermit sind Reentrants möglich

2.3.2 Beispiel

Eine Wrapperklasse die Reader Threads und Writer Threads kapselt.

```
class WithRWLock {
    final ReadWriteLock rw;
    public WithRWLock(ReadWriteLock l){ rw = l; }

    public void performRead(Runnable readCommand)
        throws InterruptedException {
        rw.readLock().acquire();
        try {
            readCommand.run();
        }finally {
            rw.readLock().release();
        }
    }
    public void performWrite(...) // similar
}
```

2.4 Semaphore

Eine Semaphore hält eine bestimmte Anzahl von permits. `acquire` decrementiert diese und `release` inkrementiert sie. Wenn die permits auf 0 gesunken sind blockiert `acquire`. Eine Semaphore die mit 1 instantiiert wird, kann wie ein Mutex verwendet werden.

Verschiedene Unterklassen bieten verschiedene Scheduling Verfahren, welcher Thread das Signal zum weitermachen erhält. Die normale Semaphore bietet keine Garantie welche Threads gestartet werden, sie ist jedoch die effizienteste.

2.4.1 Unterklassen

- **WaiterPrefenceSemaphore:** bietet eine gewisse Fairness ohne jedoch direkt eine FIFO Queue zu verwenden.
- **QueuedSemaphore:** Abstrakte Klasse für folgenden Unterklassen:
 - `FIFOSemaphore`
 - `PrioritySemaphore`: Verwaltet für jede Priorität eine eigen FIFO Queue.

3 Channel

```
public interface Takable {
    public Object take() throws InterruptedException;
    public Object poll(long ms) throws InterruptedException;
}

public interface Puttable {
    public void put(Object item) throws InterruptedException;
    public boolean poll(Object item, long ms) throws InterruptedException;
}

public interface Channel extends Puttable, Takable {
    public void put(Object item) throws InterruptedException;
    public boolean poll(Object item, long ms) throws InterruptedException;
    public Object take() throws InterruptedException;
    public Object poll(long ms) throws InterruptedException;
    public peek();
}
```

Channel ist das Interface für Buffers, Queues, Pipes usw. Channel Objekte bieten die Möglichkeit andere Objekte mit `put` hineinzustecken und wieder mit `take` herauszuholen. Zusätzlich bieten sie noch die timeout Methoden `offer` und `poll`. Mit `peek` wird eine Objekte geliefert ohne es aus der Queue zu entfernen. Sobald ein Channel Objekt keine Elemente mehr enthält blockt `take`, bis ein anderer Thread wieder ein Objekt hinzufügt. Falls die Anzahl der möglichen Objekte begrenzt ist, blockiert auch `put` sobald der Channel voll ist. Channels mit begrenzter Kapazität sollten jedoch das Interface `BoundedChannel` implementieren.

Channel ist von den Interfaces `Takable` und `Puttable` abgeleitet. Damit ist eine Typtrennung zwischen Producer und Consumer möglich.

Normalerweise enthalten Channel Objekte keine `size` Methode. Die Verwendung wäre auch nicht sehr sinnvoll, da sie immer nur eine Momentaufnahme liefern könnte. Bereits nach dem Aufruf dieser Methode könnte sich die Anzahl der Objekte bereits geändert haben.

3.1 Direkte Implementierungen

3.1.1 `LinkedQueue`

Hierbei wird eine verzeigert Liste als Datenstruktur verwendet. Die Kapazität ist theoretisch unbegrenzt. Dies ist wiederum die einfachste aber auch die effizienteste Implementierung von einem Channel.

3.1.2 `WaitFreeQueue`

Diese Klasse verwendet den Algorithmus von Michael und Scott beschrieben in "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". Dabei werden nur nicht blockende Methoden verwendet.

3.2 BoundedChannel

Wie bereits oben erwähnt stellt `BoundedChannel` das Interface für begrenzte Channels dar und ist von `Channel` abgeleitet. Dieses Interface bietet nur eine weitere Methode `capacity()`, die die maximale Anzahl von Elementen in diesem Channel liefert. Wenn ein `BoundedChannel` voll ist, blockieren die `put` Aufrufe, bis ein Element entfernt wurde.

3.2.1 Implementierungen

BoundedBuffer: effiziente auf einen Array basierende Implementierung.

BoundedLinkedListQueue: eine Variante von `LinkedListQueue` mit begrenzter Kapazität.

SemaphoreControlledChannel: Abstrakte Basisklasse für Channels die mit Semaphoren die `put` und `take` Methoden synchronisieren. Bekannte Unterklassen:

- *Slot:* Buffer mit Kapazität 1.
- *BoundedPriorityQueue:* akzeptiert nur Objekte vom Typ `Comparable`. `take` liefert immer das kleinste Element in der Liste. Als Datenstruktur wird ein normaler Heap verwendet.

SynchronousChannel: Dabei handelt es sich um einen Rendezvous Channel. Jedes `put` blockiert bis das entsprechende Element mit `take` aus dem Buffer entfernt wird.

3.3 Beispiel

In folgender Klasse wird ein eigener Thread als Logger verwendet. Die anderen Threads stellen ihre Nachrichten in eine `Message Queue(msgQ)` und der Logger holt sie heraus und gibt sie auf den Terminal aus.

```
class Service { // ...
    final Channel msgQ = new LinkedList();
    public void server() throws InterruptedException {
        String status = doService();
        msgQ.put(status);
    }
    public Service() { // start background thread
        new Thread () {
            public void run() {
                try
                    for(;;)
                        System.out.println(msgQ.take());
                } catch(InterruptedException ie) {} }
        }.start() ;
    }
}
```

4 Executor

```
public interface Executor {  
    public void execute(Runnable command) throws InterruptedException;  
}
```

Executor Objekte sind eine Erweiterung zu der Thread Klasse. Sie starten Runnables in einer bestimmten Umgebung. Die Idee war die asynchrone, oder unabhängig von Aufrufer, Ausführung von Threads.

Mit der Methode `execute(Runnable r)` wird eine Runnable ausgeführt. Dabei unterscheiden sich die verschiedenen Implementierungen, wie sie das durchführen.

Die meisten Klassen verwenden eine `ThreadFactory` zur Erzeugung der Threads. Diese kann natürlich auch überschrieben werden. Dazu muß lediglich eine Klasse das Interface `ThreadFactory` implementieren. Dadurch können die neu erzeugten Threads zB einer eigenen Thread Gruppe zugeteilt werden, oder es kann die Priorität festgelegt werden.

4.1 Einfache Implementierungen

4.1.1 DirectExecutor

Startet die `run` Methode des Runnable Objekts im aktuelle Thread.

4.1.2 LockedExecutor

Ruft die `run` Methode des Objekts innerhalb eines Sync Locks im aktuellen Thread auf. Das Sync Objekt wird dem Konstruktor übergeben.

4.1.3 ThreadedExecutor

Erzeugt einen Thread und übergibt diesem das Runnable um es auszuführen.

4.2 PooledExecutor

`PooledExecutor` hält einen Pool von Threads, die darauf warten, dass ihnen ein Runnable übergeben wird. Das bedeutet es werden mehrere Threads bei der Instantiierung erzeugt und nicht erst wenn welche gebraucht werden. Es kann die maximale Anzahl von Thread definiert werden, und bereits erzeugte Threads werden recycled was die benötigten Ressourcen begrenzt.

Diese Klasse bietet zahlreiche Methoden mit denen man das Verhalten und verschiedene Eigenschaften festlegen kann. Die wichtigsten Eigenschaften sind:

Queueing Es ist möglich eine Queue(zB `BoundedBuffer`) zu benutzen um neue Runnables zu verwalten. Dabei sollte dies nur gemacht werden wenn die einzelnen Threads unabhängig sind, da sonst Deadlocks auftreten könnten.

Maximum Pool size Default: infinite (dh `Integer.MAX_VALUE`) Es kann festgelegt werden wieviele Threads maximal erzeugt werden. Dies sollte dem Konstruktor übergeben werden.

Minimum Pool size Default: 1 bestimmt die Anzahl der Threads die mindestens laufen sollten. Wenn ein neue Anfrage kommt und weniger Threads laufen, wird sofort ein neuer erzeugt der die Anfrage übernimmt.

Preallocation mit der Methode `createThreads` kann eine bestimmte Anzahl von Threads erzeugt werden.

Keep-alive time Default: 1 min. Definiert die Zeit wie lange ein Thread mindestens im idle Zustand verbleibt bis er vom GarbageCollector gelöscht wird. Um Threads für immer zu behalten kann man einen negativen Wert verwenden.

Blocked execution policy Hiermit kann festgelegt werden was passieren soll, falls der Pool und die Queue (bei `BoundedChannel`) voll ist.

- Run (default): Der Thread der `execute` aufruft, startet die `run` Methode selber
- Wait: warten bis ein Thread frei wird. Dh blocken
- Abort: wirft eine `RuntimeException`
- Discard: verwirft die Anfrage
- DiscardOldest:

4.2.1 Beispiel

Ein Webservice mit einer maximalen Anzahl von 20 Threads. Beim Start werden 4 Threads erzeugt, und mindestens 5 Threads sollten immer laufen. Wenn zuviele Anfragen anstehen wird die älteste verworfen. Die Threads sollten mindestens 5 Minuten unbeschäftigt sein, bevor sie gelöscht werden können.

```
class WebService {
    public static void main(String[] args) {
        PooledExecutor pool =
            new PooledExecutor(new BoundedBuffer(10), 20);
        pool.createThreads(4);
        pool.setMinimumPoolSize(5);
        pool.discardOldestWhenBlocked();
        pool.setKeepAliveTime(1000 * 60 * 5);

        try {
            ServerSocket socket = new ServerSocket(9999);
            for(;;) {
                final Socket connection = socket.accept();
                pool.execute(new Runnable() {
                    public void run() {
                        new Handler().process(connection);
                    }
                });
            }
        } catch (Exception ex) { }
    }
}
```

5 Barrier

```
public interface Barrier {
    public int parties();
    public boolean broken();
}
```

Barriers sind Synchronisationspunkte für Threads die aufeinander warten sollten oder müssen. Barriers unterstützen dies mit unterschiedlichen Methoden. Dieses Interface bietet nur die minimalen Gemeinsamkeiten:

- Jede Barrier ist für eine bestimmte Anzahl von `parties` definiert. Dies ist eine Konstante die bei der Konstruktion des Barriers angegeben werden muß.
- Eine Barrier kann `broken` sein. Das bedeutet normalerweise, dass ein Thread die Barrier vorzeitig verlassen hat. Danach werden alle synchronisations Aufrufe zu dieser Barrier mit `BrokenBarrierException` abgebrochen.

5.1 CyclicBarrier

Ein `CyclicBarrier` ist ein Punkt an dem eine fixe Anzahl von Threads synchronisiert werden. Es findet zwischen den Threads kein Datenaustausch statt, es kann jedoch optional ein `Runnable` angegeben werden, das jedesmal ausgeführt wird wenn die Barrier freigegeben wird, dh sobald alle Threads `Barrier.barrier()` aufgerufen haben.

wichtige Methoden:

barrier(): auf die anderen Threads warten

attemptBarrier(long ms): wartet max ms Millisekunden auf die anderen Threads

Der Rückgabewert ist bei beiden Methoden der ankunfts Index. Falls der Rückgabewert 0 ist, war dies der letzte Thread und er war verantwortlich dafür die optionale Runnable auszuführen und die anderen zu wecken.

5.2 Rendezvous

Rendezvous sind Synchronisationspunkte für Threads die Daten austauschen. Diese Barrier kann für weniger Threads definiert werden als daran beteiligt sind, aber es werden immer nur so viele freigegeben wie definiert wurde. Die Funktion wie die Daten getauscht werden, kann selbst definiert werden, default wird eine einfache Rotation vorgenommen.

wichtige Methoden:

Object rendezvous(Object x): auf die anderen Threads warten, und das Objekt x übergeben. Der Rückgabewert ist ein anderes Object von einem anderen Thread.

Object attemptRendezvous(Object x, long ms): wartet max ms Millisekunden auf die anderen Threads, vertauscht die Objekte.

5.2.1 Rendezvous.RendezvousFunction

```
public static interface Rendezvous.RendezvousFunction {
    void rendezvousFunction(Object[] objects)
}
```

Mit Hilfe dieses Interfaces kann die default Rotationsfunktion bei Rendezvous überschrieben werden. Der `objects` Array enthält die Objekte die die einzelnen Threads dem Rendezvous mitgeben. Die Threads erhalten das Objekt, welches nach dieser Methode bei ihrem jeweiligen Index steht. zB. sollte der erste Thread mit dem dritten Thread die Elemente tauschen, so müssen nur die Objekte im Array vertauscht werden, dh das Objekt im ersten Feld des Arrays wird mit dem Dritten vertauscht.

6 Santa Claus Problem

Folgendes Beispiel soll die Verwendung von Barriers demonstrieren. Es gibt drei Möglichkeiten zur Synchronisation von Threads:

- `CyclicBarrier` für einfaches Warten auf andere Threads
- `Rendezvous` zum bidirektionalen Datenaustausch
- `SynchronousChannel` zum Datenaustausch zwischen zwei Threads in einer Richtung

Für unser Problem benötigen wir die ersten zwei.

6.1 Problemstellung

Das Problem wurde von J.A. Trono gestellt[3]. Er hat auch eine Lösung mit Semaphoren präsentiert. Seine Lösung verwendet zehn Semaphoren und zwei globale Variablen und ist trotzdem nicht ganz korrekt. M. Ben-Ari stellte eine Lösung mit ADA95 und normalen Java synchronisations Methoden vor[1].

Santa Claus schläft am Nordpol in seinem Büro bis er entweder von allen Rentieren oder von einer Gruppe von drei Elven geweckt wird. In Abhängigkeit von wem er geweckt wurde muß er unterschiedliche Arbeiten verrichten:

- wenn ihn die Rentiere wecken, muß er sie vor den Schlitten spannen, gemeinsam das Spielzeug verteilen, und danach wieder ausspannen.
- wenn ihn eine Gruppe von Elven weckt, bittet er sie ins Büro, bespricht ihre Probleme und führt sie wieder hinaus.

Das Zusammenstellen von den Gruppen wird nicht von Santa erledigt, da seine Zeit sehr kostbar ist. Die wartenden Reentiere sollten vor den Elven abgefertigt werden.

6.2 Implementierung

Das Verhalten von den Elven und Reentieren ist analog. Im Folgenden werde ich auf die Reentier-Threads eingehen.

6.2.1 Gruppenbildung

Die einzelnen Reentiere müssen auf den Rest der Gruppe warten. Ein Reentiere muß danach Santa wecken und ihm die wartende Gruppe übergeben. Zur Gruppenbildung verwende ich ein `Rendezvous`, instantiiert für alle Reentiere im Unterschied zu den Elven, bei denen wird das `Rendezvous` mit 3 instantiiert. Dieses `Rendezvous` wird den Rentieren bei der Konstruktion übergeben. Jedes Reentier gibt dem `Rendezvous` die Referenz zu sich selbst mit, aber nur eines bekommt ein `Group` Objekt zurück welches alle Rentiere enthält, die anderen erhalten `null` zurück.

Um die letzte Bedingung zu erfüllen muß die `Group` Klasse das Interface `Comparable` implementieren. Santa verarbeitet dann immer die Gruppe mit der niedrigsten `GroupID`.

6.2.2 Santa wecken

Santa schläft und wartet bis ihm eine komplette Gruppe Elven oder Reentiere übergeben wird. Dazu verwende ich eine `BoundedPriorityQueue` im Santa Objekt als `wake`. Das Reentier welches das `Group` Objekt erhalten hat, setzt die richtige `GroupID` damit Santa erkennt was zu tun ist und gibt es in die Queue.

6.2.3 ein-/ausspannen

Nach dem Santa geweckt wurde, wartet jedes Reentier bis es von Santa vor den Schlitten gespannt wird. Nach dem die Geschenke ausgeliefert wurden, warten sie bis Santa sie wieder ausspannt. Dies wird mit einfachen `CyclicBarrier` gemacht. Jedes Reentier Objekt enthält dazu zwei `CyclicBarrier`, `harness` und `unharness`, die Santa bei jeden Reentier der Reihe nach freigibt.

6.3 Code

```
import EDU.oswego.cs.dl.util.concurrent.*;

public class Santa
{
    private static final int numReindeer = 9;
    private static final int numElve = 16;
    private static final int minElve = 3;

    private static final int ReindeerID = 0;
    private static final int ElveID = 1;

    Channel msgQ; // Queue for output

    private final Claus santa; // every Santa objects has his Claus

    Santa() // constructor
    {
        msgQ = new LinkedQueue();
    }

    class Group
    {
        public int GroupID;
        public Object[] members;

        public int compareTo(Object o) implements Comparable

        Group other = (Group)o;
        if(GroupID == other.GroupID)
            return 0;
        else if(GroupID < other.GroupID)
            return -1;
        else
            return 1;
    }

    class MyRendezvousFunction
```

```
        implements Rendezvous.RendezvousFunction
    {
        public void rendezvousFunction(Object[] objects)
        {
            Group group = new Group();
            group.members = (Object[])objects.clone();

            /* one of the waiting Threads get the reference to all other Objects */
            objects[0] = group;
            for(int i = 1; i < objects.length; i++)
                objects[i] = null ;
        }
    }

class Reindeer implements Runnable
{
    public String name;
    Rendezvous wait_other;
    CyclicBarrier harness, unharness;
    Group others;

    Reindeer(String name, Rendezvous r)
    {
        this.name = name;
        wait_other = r;
        harness = new CyclicBarrier(2);
        unharness = new CyclicBarrier(2);
    }

    public void run()
    {
        for(;;)
        {
            try{
                /* on vacation */
                msgQ.put("Reindeer " + name + " on vacation ");
                Thread.sleep((long) (Math.random()*1000));
                others = (Group)(wait_other.rendezvous(this)); //wait for others
                if(others != null) // one of Group has to wake Santa
                {
                    others.GroupID = ReindeerID;
                    /*Reindeers waking Santa */
                    santa.wake.put(others);
                    System.out.println(" Reindeer wated " + ms);
                }
            }
            try{
                harness.barrier(); // wait for Santa harness me
            } catch (BrokenBarrierException ex) { return; }
            /* Deliver Toys */
            msgQ.put("Reindeer " + name + " deliver toys");
        }
    }
}
```

```

        try{
            unharness.barrier(); // wait for Santa unharness me
        } catch (BrokenBarrierException ex) { return; }

    } catch (InterruptedException es) { return; }
    }
}

class Elve implements Runnable
{
    public String name;
    Rendezvous wait_other;
    CyclicBarrier invite_in, show_out;
    Group others;

    Elve(String name, Rendezvous r)
    {
        this.name = name;
        wait_other = r;
        invite_in = new CyclicBarrier(2);
        show_out = new CyclicBarrier(2);
    }

    public void run()
    {
        for(;;)
        {
            try{
                msgQ.put("Elve " + name + " working" );
                Thread.sleep((long) (Math.random()*1000));
                others = (Group)(wait_other.rendezvous(this)); //wait for others
                if(others != null) // one of Group has to wake Santa
                {
                    others.GroupID = ElveID;
                    /* Elves waking Santa */
                    santa.wake.put(others);
                }
            } catch (BrokenBarrierException ex) { return; }
            /* Consulting with Santa */
            msgQ.put("Elve " + name + " consulting with Santa");
            try{
                show_out.barrier(); // waiting for Santa show me out
            } catch (BrokenBarrierException ex) { return; }
        } catch (InterruptedException ex) { return; }
    }
}

```

```
}

class Claus implements Runnable
{
    /* point for Elves or Reindeer to wake me */
    public final BoundedPriorityQueue wake;
    Group group;

    Claus(){
        wake = new BoundedPriorityQueue(10);
    }

    public void run()
    {
        for(;;)
        {
            /* sleeping */
            try{
                msgQ.put("Santa is sleeping");
                group = (Group)( wake.take() );
            } catch (InterruptedException es){ return; }

            if( group.GroupID == ReindeerID ) // check who is waking me
            try{
                msgQ.put("Santa was waked from Reindeers");

                /* harness all Reindeer */
                for(int i = 0; i < group.members.length; i++)
                {
                    msgQ.put("Santa harness Reindeer " +
                        ((Reindeer)group.members[i]).name );
                    try{
                        ((Reindeer)group.members[i] ).harness.barrier();
                    } catch (BrokenBarrierException ex){ return; }
                }
                /* deliver toys */
                msgQ.put("Santa deliver toys");
                Thread.sleep(1000);
                /* unharness all Reindeers */
                for(int i = 0; i < group.members.length; i++)
                try{
                    ((Reindeer)group.members[i] ).unharness.barrier();
                    msgQ.put("Santa unharness Reindeer " +
                        ((Reindeer)group.members[i]).name );
                } catch (BrokenBarrierException ex){ return; }
            } catch (InterruptedException ex ) { return; }

            else // Elves
            try{
                msgQ.put("Santa was waked from Elves");
            }
        }
    }
}
```



```
        ).start();
    }
    public static void main(String[] args)
    { new Santa().start(); }
}
```

7 Zusammenfassung

Das Paket `util.concurrent` ist einfach zu verwenden. Die Dokumentation der API ist sehr gut, dank JavaDoc. Es sind sehr viele Beispiele in der Dokumentation, die die Verwendung demonstrieren. Das Paket ist eine nützliche Erweiterung zu den in Java integrierten Synchronisationsmethoden. Da es wahrscheinlich in zukünftigen Java Releases enthalten sein wird, ist es durchaus empfehlenswert dieses Paket zu verwenden.

Das Paket enthält noch weitere Klassen die hier nicht erwähnt wurden. So wurde der Großteil der Collections die in Java vorhanden sind, neu implementiert. Zusätzlich wurden noch Fork/Join Tasks für die Parallelverarbeitung implementiert.

Literatur

- [1] Mordechai Ben-Ari. How to solve the santa claus problem. *Concurrency: Practice & Experience*, 10(6):485–496, 1998. <http://stwi.weizmann.ac.il/g-cs/benari/articles/santa.pdf>. 15
- [2] Doug Lea. `util.concurrent`. <http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html>. 3
- [3] John A. Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26(3):8–10, 1994. <http://academics.smcvt.edu/jtrono/Papers/SANTA.DOC>. 15