Freie Universität Berlin

Prof. Dr. Margarita Esponda

Nichtsequentielle Programmierung, SoeSe 2017

Übungsblatt 5

TutorIn: Lilli Walter
Tutorium 6

Boyan Hristov, Sergelen Gongor

13. Juni 2017

Link zum Git Repository: https://github.com/BoyanH/FU-Berlin-ALP4/tree/master/Solutions/Homework5

Aufgabe 1

1. $\Box A \wedge \Diamond B \Rightarrow \Diamond(A \wedge B)$

$\Box A \wedge \Diamond B$
$\Leftrightarrow \forall k \geq i : A \wedge \exists j \geq i : B$
$\Leftrightarrow \exists j \geq i : A \wedge B$ (Da A für alle k $\geq$ i wahr ist, dann auch für j)
$\Leftrightarrow \Diamond(A \wedge B)$

2. $\Box(A \vee B) \Rightarrow (\Box A \vee \Diamond B)$

$\Box(A \vee B)$

$\Leftrightarrow \forall k \geq i : A \vee B$
Sei Z die Menge aller $p \geq i$ mit A wahr in S_p, dann gilt
$\forall z \in Z : A \wedge \forall q \geq i, q \notin Z : B$
$\Rightarrow \Box A \vee \Box B \vee \Diamond A \vee \Diamond B$
$\Rightarrow (\Box A \vee \Diamond B)$

Die Formale Begründung war schwierig und uns reichte die formale Notation nicht aus, aber informal:
Das linke Teil der Implikation sagt, dass in jedem Zustand gilt mindestens eine von den beiden A und B. Dann muss entweder eine von den beiden immer gelten, oder wenn es mindestens ein

Zustand gibt, wo das eine nicht gilt, muss das andere in diesem Zustand gelten. D.h. wenn A nicht immer wahr ist, must B irgendwann wahr sein und umgekehrt.

Da aber $\Box B \Rightarrow \Diamond B$, oder wenn B immer gilt, dann gilt irgendwann B, kann man aus dem linken Teil der Implikation den rechten ableiten, oder anders gesagt, wenn das linke wahr ist, ist das rechte Teil auch immer wahr.

3. $\Diamond A \wedge \Diamond B \Rightarrow \Diamond(A \wedge B)$

   Diese Aussage ist falsch. Gegenbeispiel:

   Sei A wahr in S_j bis S_{j+2} und B wahr von S_{j+4} bis S_{j+6}, dann gilt$\Diamond A \wedge \Diamond B$ aber nicht $\Diamond(A \wedge B)$

   Informal: Das linke Teil sagt, dass jede Aussage in mindestens einem beliebigen Zustand wahr ist, aber nicht das beide Aussagen in einem beliebigen Zustand wahr sind, was das rechte Teil ist. In der Vorlesung wurde $\Box\Diamond A \wedge \Box\Diamond B \not\Rightarrow \Diamond(A \wedge B)$ gezeigt, was eigentlich eine Obermenge von dieser Aufgabe ist. Da haben wir gesehen, dass auch wenn die zwei Aussagen ständig wieder wahr werden, gibt es keine Garantie, dass diese irgendwann zusammen beide gelten werden.

Aufgabe 2

1. n=4, k=2

   Die erste zwei Threads laufen bis p_8 nach einander. D.h. $count = 0 \wedge D = 0$. Dan laufen zwei weitere Threads nach einander bis p_7. D.h. $count = -2$ und die letzte beide Threads warten auf D. Danach laufen die erste zwei Threads bis p_1. Das erste Thread setzt $count = count + 1 = -2 + 1 = -1 \leq 0$ und released deswegen D. Dann wird das 2 Thread, das sich im CS befindet bis p_1 ausgeführt, setzt dabei $count = count + 1 = -1 + 1 = 0 \leq 0 \Rightarrow$ releaset D wieder. Damit haben wir $D = 2$ erreicht, ein unerlaubtes Zustand.

2. n=3, k=2

   T_1 und T_2 laufen bis p_8, damit ist count = 0, D = 0. T_3 läuft bis p_7 und wartet auf D, count = -1. T_1 läuft bis p_7, setzt dabei count = 0 in $p_1 0$ und wieder count = -1 in p_3, deswegen geht er in die if-Anweisung. D wird von T_1 dabei release-t und es gilt jetzt D = 1. T_2 macht das selbe, da count = -1 setzt dieses count = 0 in $p_1 0$ und wieder $count = 3$ in p_3. Dabei release-t er aber auch D, da count = 0 war als er durchgelaufen ist, deswegen ist jetzt D = 2, ein unerlaubtes Zustand.

   Das kann passieren nur wenn T_1 und T_2 in p_6 oder p_7 lang genug hängen bleiben, wenn der Scheduler diese nicht laufen lässt, diese aber warten seit dem T_1 $p_1 2$ ausgeführt hat nicht mehr auf acquire(D)!!!

Aufgabe 3

Unsere Lösung funktioniert mit den Invarianten:

1. Nur 3 Threads werden durch den Barrier auf einmal durchgelassen (durch Java garantiert), maximal 3 warten

   Formal: barrier.getParties() $= 3 \wedge$ barrier.getNumberWaiting() ≥ 3

2. Nur 2 Wasserstoff und 1 Sauerstoff Atome können gleichzeitig an der Barriere warten. Wird durch den Semaphoren garantiert.

   Formal: Hydrogen.hydrogens.availablePermits() $\geq 2 \wedge$ Oxygen.oxygens.availablePermits() ≥ 1

```java
package fu.alp4;

import java.util.concurrent.CyclicBarrier;

public class Main {

    public static void main(String[] args) {
        final CyclicBarrier barrier = new CyclicBarrier(3);

        while (true) {
            double random = Math.random();

            if (random > 0.4) {
                new Hydrogen(barrier).start();
            } else {
                new Oxygen(barrier).start();
            }

            Nap.randomNap(1000, 3000);
        }

    }
}
```

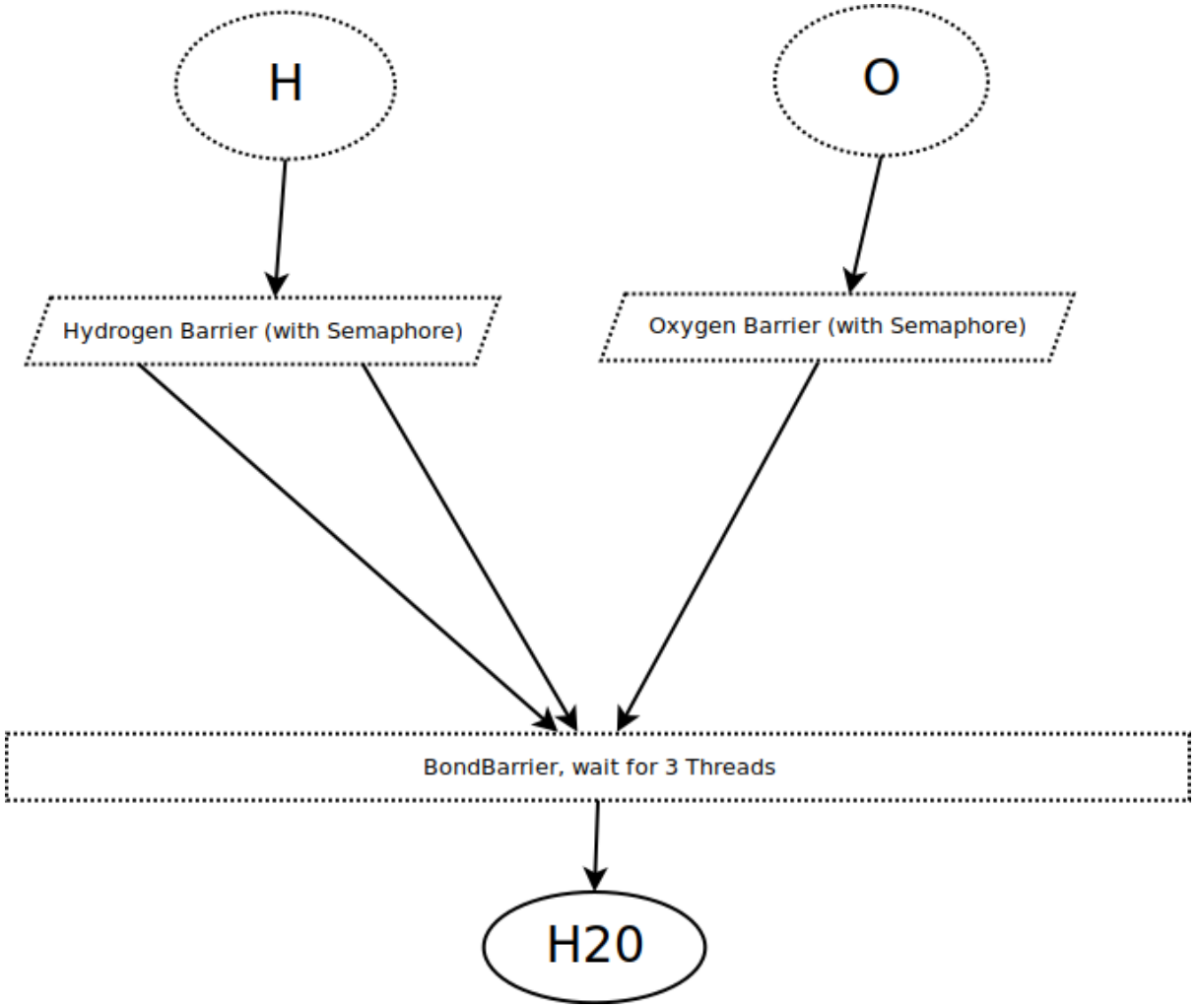
```java
package fu.alp4;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public abstract class BarrierReleaseHandler extends Nap {

    static int waterMolecules = 0;
    private Lock incrementLock = new ReentrantLock();
    private CyclicBarrier barrier;

    public BarrierReleaseHandler(CyclicBarrier bondingBarrier) {
        this.barrier = bondingBarrier;
    }

    protected void awaitBarrier() throws BrokenBarrierException, InterruptedException {
        int turn = this.barrier.await();

        /**
         * The first thread to arrive at the barrier will be the one to create the water molecule.
         * Method created to share functionality between Hydrogen and Oxygen classes
         */

        if (turn == 0) {
            incrementLock.lock();
            BarrierReleaseHandler.waterMolecules++;
```

```
            System.out.printf("A new water molecule was created! Water molecules: %d\n",
    BarrierReleaseHandler.waterMolecules);
            incrementLock.unlock();
        }
    }
}
```

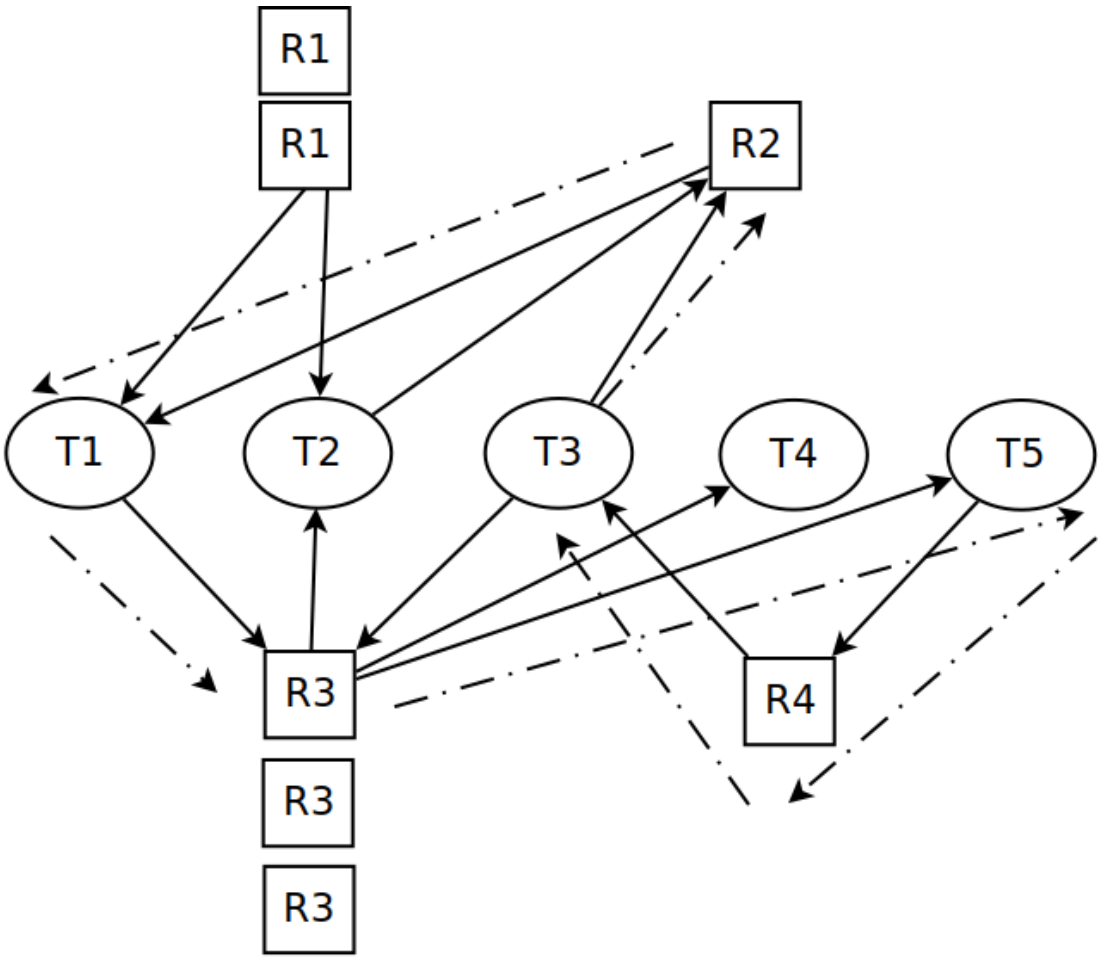
```
package fu.alp4;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Semaphore;

public class Hydrogen extends BarrierReleaseHandler {

    static final Semaphore hydrogens = new Semaphore(2);

    public Hydrogen (CyclicBarrier bondingBarrier) {
        super(bondingBarrier);
    }

    public void run() {

        System.out.println("Hydrogen atom appeared from nowhere!");
        Hydrogen.randomNap(2000, 5000);
        System.out.println("Hydrogen atom wants to bond!");
        try {
            /**
             * allow only up to 2 hydrogens to wait at the barrier
             */
            this.hydrogens.acquire();
            System.out.println("Hydrogen atom is ready to bond, waiting on the barrier!");
        } catch (InterruptedException e) {
            e.printStackTrace();
            return;
        }

        try {
            this.awaitBarrier();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }

        // release back the available spots after molecule bonding
        hydrogens.release();
    }
}
```

```
package fu.alp4;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Semaphore;

public class Oxygen extends BarrierReleaseHandler {

    static final Semaphore oxygens = new Semaphore(1);
    private CyclicBarrier barrier;

    public Oxygen(CyclicBarrier bondingBarrier) {
        super(bondingBarrier);
    }
```

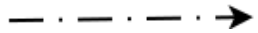
```
17      public void run() {
18
19          System.out.println("Oxygen atom appeared from nowhere!");
20          Hydrogen.randomNap(2000, 5000);
21          System.out.println("Oxygen atom wants to bond!");
22          try {
23              this.oxygens.acquire();
24              System.out.println("Oxygen atom is ready to bond, waiting on the barrier!");
25          } catch (InterruptedException e) {
26              e.printStackTrace();
27              return;
28          }
29
30          try {
31              this.awaitBarrier();
32          } catch (InterruptedException e) {
33              e.printStackTrace();
34          } catch (BrokenBarrierException e) {
35              e.printStackTrace();
36          }
37
38          oxygens.release();
39      }
40  }
```



Aufgabe 4



Ein Kreis in dem Graph



Ein Kreis haben wir auf dem Graphen dargestellt, es existiert aber noch ein weiteres und nämlich $T_2 \to R_2 \to T_1 \to R_3 \to T_2$.

1. Der Graph befindet sich in einem von Deadlock gefährdeten Zustand, da es Zyklen in dem Graph gibt
2. Da es keine Kreis / Zyklus / Schleife zu finden ist, wo nur einzelne Resourcen ($R_i = 1 \forall i \in R$), können wir nicht mit Sicherheit sagen, dass es Deadlock gibt. Nach unserer Beobachtung haben wir keine gefunden.

Aufgabe 5

Da es maximal n+m Resourcen angefordert werden müssen, heißt es auch in dem Fall, dass ein Thread extrem Resourcenaufwändig ist und n-1 Resourcen schon genommen hat und n allgemein braucht, aber

nicht das letzte sich geholt hat damit es terminiert und die Resourcen wieder freigibt, gibt es immer noch m-2 Threads die nur eine Resource brauchen und frei funktionieren können.

Wie wir schon kennen, ist es aber am gefährlichsten wenn alle Threads ungefähr gleich viele Resourcen brauchen. Sei k diese Anzahl, dann ist die gefährlichste Situation, wenn jeder Thread k-1 Resourcen sich geholt hat, aber das letzte noch nicht und terminiert deswegen nicht, gibt auch keine Resourcen noch frei. In diesem Fall sind das $\lfloor \frac{(n+m-1)}{m} \rfloor$ Resourcen pro Prozess. Also jedes Prozess nimmt initial $\lfloor \frac{(n+m-1)}{m} - 1 \rfloor = \lfloor \frac{n}{m} + 1 - \frac{1}{m} \rfloor$ Resourcen. Da das keine ganze Zahl (wegen $\frac{1}{m}$) ist bleibt immer ein (genau 1 wegen $m\frac{1}{m}$) Resource frei / bzw. ein Thread braucht wenigere Resourcen und kann terminieren und die Resourcen wieder frei lassen.