2022

Parallele und Verteilte Systeme

Nebenläufige Programme in Java – Gruppe C1

DENNIS JONGEBLOED, 7010939
ABDULLATIF ZANABILI, 7014798
RAINER PEDDE, 7014109
DENNIS SEILER, 7011776

Inhaltsverzeichnis

Glossar

Kritischer Abschnitt

Kritische Abschnitte sind Bereiche, auf die mindestens zwei konkurrente Threads oder Prozesse zugreifen möchten, um gemeinsame Daten lesend und/oder schreibend zu verarbeiten.

Mutual Exclusion/Wechselseitiger Ausschluss

Die Mutual Exclusion ist auch als wechselseitiger Ausschluss bekannt. Sie hat die Aufgabe, dass gleichzeitige Zugreifen von mehreren Prozessen auf die selben geteilten Variablen zu unterbinden.

Shared Memory/geteilte Variable

Shared Memory wird genutzt, damit mehrere Objekte ein und denselben Speicher bzw. Variable nutzen können und somit ohne direkte Übergabe Daten überreichen oder gemeinsam nutzen und verändern können. Die Nutzung von Shared Memory/geteilten Variablen ist Teil eines kritischen Abschnitts.

Aufgabe 1: Nebenläufige Programme in Java

In der ersten Aufgabe des Parallele und Verteilte Systeme Praktikums haben wir ein Ampelsystem für eine Straßenkreuzung in Java implementiert. Jede Ampel wird durch eine Himmelsrichtung (Enumeration CardinalDirection) identifiziert, d.h. es gibt insgesamt vier Ampeln (NORTH, SOUTH, EAST und WEST).

Zudem werden die Farben der Ampel mit der Enumeration Colour festgelegt (RED, YELLOW, GREEN). Jede Ampel beginnt mit dem Zustand RED und wechselt die Farben in der Reihenfolge: RED --> GREEN --> YELLOW --> RED --> GREEN usw. Alle Ampeln sollen als unabhängige und nebenläufige Threads arbeiten.

Die Klasse TrafficLight erbt von der Klasse Thread, da wir die Funktionen von Threads benötigen.

```
public class TrafficLight extends Thread
```

Klasse Intersection(Main)

In der Klasse Intersection befindet sich die Main-Methode. Hier werden die 4 Threads der Klasse TrafficLight erstellt und über die Methode start() aktiviert. Zudem wird die Ampelschaltung nach 50 Millisekunden angehalten.

```
public class Intersection {
   public static void main(String[] args) {
       CardinalDirection startDir = CardinalDirection.NORTH;

       TrafficLight lightEast = new TrafficLight(CardinalDirection.EAST,
startDir);
       lightEast.start();

       TrafficLight lightNorth = new TrafficLight(CardinalDirection.NORTH,
startDir);
       lightNorth.start();

       TrafficLight lightSouth = new TrafficLight(CardinalDirection.SOUTH,
startDir);
       lightSouth.start();

       TrafficLight lightWest = new TrafficLight(CardinalDirection.WEST,
startDir);
       lightWest.start();

   try {
         Thread.sleep(50);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
       lightEast.halt();
    }
  }
}
```

Klasse TrafficLight

Die Ampeln besitzen eine CardinalDirection (cd) um den Standort der Ampel zu spezifizieren und eine weitere CardinalDirection (dir) um festzulegen, welche Ampel-Himmelsrichtung zuerst schalten darf. Zum Beispiel, wenn 'dir' den Wert 'NORTH' besitzt, dürfen die Ampeln der Himmelsrichtungen 'NORTH' und 'SOUTH' zuerst in die Grünphase schalten. Die Variable (dir) wurde hier als Shared Variable festgelegt.

```
private CardinalDirection cd;
private static volatile CardinalDirection dir;
```

Die Variable 'color' beschreibt die Farbe der Ampel und ist für jedes Objekt separat, der Default Wert ist RED.

```
private Colour color;
```

'nextColor' dient zum Erkennen, welche Ampelfarbe für die nächste Ampelphase benötigt wird.

```
private static volatile Colour nextColor;
```

Um die Ampelschaltung wieder zu stoppen, wurde ein geteilter Boolean 'stopped' erstellt.

```
private static volatile boolean stopped = false;
```

'mainReady' und 'oppReady' sind geteilte Boolean, sie speichern, ob die zu schaltenden Richtungen die Farbe der aktuellen Ampelphase angenommen haben.

```
private static volatile boolean mainReady = false;
private static volatile boolean oppReady = false;
```

Das Lock Object dient für Synchronized als einmalige Referenz, sodass nur ein Thread die damit markierten Bereiche nutzen kann.

```
private static final Object lock = new Object();
```

Der Konstruktor

Der Konstruktor besitzt als Übergabeparameter zweimal die Enumeration 'CardinalDirection'. Der Parameter 'cd' ist der Standort der Ampel und 'dir' ist die Ampel-Startrichtung, welche starten darf.

Zudem wird 'color' auf RED gesetzt, die 'nextColor' Variable wird auf GREEN gesetzt.

```
 public TrafficLight(CardinalDirection cd, CardinalDirection dir) {
    color = Colour.RED;
    nextColor = Colour.GREEN;
    this.cd = cd;
    TrafficLight.dir = dir;
  }
```

Die Run () Methode

```
  @Override
  public void run() {

    Reporter.show(cd, color);

    while (!stopped) {
      if (cd == dir || cd == CardinalDirection.opposite(dir)) {
        if (color != nextColor) { // If current colour is not equal to next
(expected) colour
          synchronized (lock) { // mutual exclusion (critical area)

            if (cd == dir) {
```

```
              mainReady = true;
              color = Colour.next(color); // Switch current traffic light to the
next colour
              Reporter.show(cd, color);
            } else if (cd == CardinalDirection.opposite(dir)) {
              oppReady = true;
              color = Colour.next(color); // Switch current traffic light to the
next colour
              Reporter.show(cd, color);
            }
          }

          synchronized (lock) { // mutual exclusion (critical area)
            if (mainReady && oppReady) {
              nextColor = Colour.next(nextColor); // Next (expected) traffic light
colour
              mainReady = false;
              oppReady = false;
              if (nextColor == Colour.GREEN) { // If expected colour is green, the
Axis will switch
                dir = CardinalDirection.next(cd);
              }
            }
          }
        }
      }
    }
  }
```

Die While-Schleife wird solange ausgeführt, bis die Variable auf 'false' gesetzt wird. Wenn dies passiert, ist die Ampelschaltung gestoppt.

```
while (!stopped)
```

Diese Abfrage überprüft, welche Ampel-Himmelrichtung geschaltet werden darf. Als Beispiel, wenn 'dir' auf EAST gesetzt ist, dürfen die Ampeln mit der Richtung 'EAST' und die gegenüberliegende Ampel 'WEST' schalten.

```
if (cd == dir || cd == CardinalDirection.opposite(dir))
```

Hier wird überprüft, ob die Farbe der Ampel nicht mit der erwarteten Farbe übereinstimmt, also noch nicht weiter geschaltet wurde.

```
if (color != nextColor)
```

Der Ausdruck 'synchronized' dient für die Sicherung der Datenkonsistenz, somit kann immer nur ein Thread den kritischen Abschnitt betreten. Sobald die Variable 'dir' mit der Variable 'cd' übereinstimmt, wird 'mainReady' auf true gesetzt. Falls 'cd' mit der entgegengesetzten Richtung von 'dir' übereinstimmt, sind wir im Fall der gegenüberliegenden Ampel und somit wird 'oppReady' auf true gesetzt. In der nächsten Zeile wird die Farbe der Ampel auf die nächste Farbe gesetzt. Die Reihenfolge sieht wie folgt aus: RED -> GREEN -> YELLOW -> RED...

Reporter.show dient zur Ausgabe der Ampel-Himmelsrichtung und die aktuelle Farbe der Ampel. Beispielausgabe: "NORTH: GREEN"

```
  synchronized (lock) { // mutual exclusion (critical area)
    if (cd == dir) {
      mainReady = true;
      color = Colour.next(color); // Switch current traffic light to the next
colour
      Reporter.show(cd, color);
    } else if (cd == CardinalDirection.opposite(dir)) {
      oppReady = true;
      color = Colour.next(color); // Switch current traffic light to the next
colour
      Reporter.show(cd, color);
    }
  }
```

In diesem Synchronized Abschnitt wird erst überprüft, ob beide Ampeln einer Himmelsrichtung geschaltet wurden. Ist dies der Fall, wird 'nextcolor' auf die nächste Farbe geändert. Anschließend werden 'mainReady' und 'oppReady' für die nächste Ampelphase auf false gesetzt. Bei der zweiten Abfrage wird überprüft, ob 'nextColor' erneut den Wert GREEN angenommen hat. Wenn dies der Fall ist, wird die Himmelsrichtung verändert.

```
  synchronized (lock) { // mutual exclusion (critical area)
    if (mainReady && oppReady) {
      nextColor = Colour.next(nextColor); // Next (expected) traffic light colour
      mainReady = false;
      oppReady = false;
      if (nextColor == Colour.GREEN) { // If expected colour is green, the Axis
will switch
        dir = CardinalDirection.next(cd);
      }
    }
  }
```

Die halt() Methode

Beim Aufruf dieser Methode wird die Variable 'stopped' auf true gesetzt und die Ampelschaltung gestoppt.

```
  public void halt() {
    stopped = true;
  }
```

Hier läuft das Programm wie folgt ab, ausführlichere Ausgaben liegen der .zip Datei bei:

```
SOUTH: GREEN
NORTH: GREEN
NORTH: YELLOW
SOUTH: YELLOW
```

```
SOUTH: RED
NORTH: RED
WEST: GREEN
EAST: GREEN
WEST: YELLOW
EAST: YELLOW
EAST: RED
WEST: RED
NORTH: GREEN
SOUTH: GREEN
SOUTH: YELLOW
NORTH: YELLOW
NORTH: RED
SOUTH: RED
////----------\\
```