ParaProg Übung W04, Testat 01

Parallele Prgrammierung

Emanuel Duss

2014-03-12 18:14

Inhaltsverzeichnis

1 Ziele

- Problematik der Race Conditions vertiefen und analysieren.
- Deadlock- und Starvation-Probleme erkennen und beheben.
- Faire erweiterte Synchronisationsprimitive selber entwickeln.

2 Aufgabe 1: Broken Cyclic Barrier (Theorie)

2.1 Problemstellung

Folgender Ansatz versucht vergeblich eine zyklische Synchronisation mit einem CountDownLatch zu realisieren (vollständiger Code in der Vorlage). Bestimmen Sie den Nebenläufigkeitsfehler.

```
CountDownLatch latch = new CountDownLatch(NOF_THREADS);
  void multiRounds(int threadId) throws InterruptedException {
    for (int round = 0; round < NOF_ROUNDS; round++) {
      latch.countDown();
      latch.await();
      if (threadId == 0) {
        latch = new CountDownLatch(NOF_THREADS); // new latch for new round
      }
      System.out.println("Round " + round + " thread " + threadId);
  }
}
```

2.2 Problemanalyse

In der Dokumentation zu `Class DountDownLatch` steht:

> This is a one-shot phenomenon – the count cannot be reset. If you need a version that resets the count, consider using a CyclicBarrier.

In der Dokumentation zu `Class CyclicBarrier` steht:

> The barrier is called cyclic because it can be re-used after the waiting threads are released.

Somit soll das Problem mit einer `CyclicBarrier` umgesetzt werden:

Das wirft aber eine `BrokenBarrierException`. Es existiert also weiterhin einn Problem. Anmekrung zu Das zurücksetzen des Counters bei einem `CountDownLatch` ist mit einer neuzuweisung gültig.

2.3 Problembeschreibung

- Es ist nicht klar, wann der `Thread-0` den Counter zurücksetzt.
- Eventuell beginnen die anderen Threads die neue Runde, bevor `Thread-0` den Counter zurückgesetzt hat.

2.4 Problemlösung

Idee Nummer 1: Das ist mit einem zweiten zweiten `CountDownLatch` lösbar:

```
  private static void multiRounds(int threadId) throws InterruptedException {
    for (int round = 0; round < NOF_ROUNDS; round++) {
      if (round % 2 == 0){
        latch.countDown();
        latch.await();
```

```
        if (threadId == 0)
          latch = new CountDownLatch(NOF_THREADS); // new latch for new round
      }
      else {
        latch2.countDown();
        latch2.await();
        if (threadId == 0)
          latch2 = new CountDownLatch(NOF_THREADS); // new latch for new round
      }
      System.out.println("Round " + round + " thread " + threadId);
    }
  }
```

Jetzt wird bei jeder Runde zwischen zwei `CountDownLatch` abgewechselt. Somit wird bei einer neuen Runde nich der "alte" `CountDownLatch` verwendet, sondern ein anderer (abwechslungsweise).

Das hat jedoch nicht funktioniert. Nach mehrmaligem ausführen blockierten die Threads weiter. Es findet also weiterhin eine Race Condition statt. Das Problem tritt auf, wenn ein Thread ein `countDown()` ausführt aber nicht zu `await()` kommt. In der Zwischenzeit kann der `Thread-0` den Counter zurücksetzen. Erst jetzt kommt der andere Thread zum `await()` und wartet für die "spätere" runde.

```
private static void multiRounds(int threadId) throws InterruptedException {
    for (int round = 0; round < NOF_ROUNDS; round++) {
      if (round % 3 == 0) {
        latch1.countDown();
        latch1.await();
        if (threadId == 0)
          latch3 = new CountDownLatch(NOF_THREADS); // new latch for new round
      } else if (round %3 == 1) {
        latch2.countDown();
        latch2.await();
        if (threadId == 0)
          latch1 = new CountDownLatch(NOF_THREADS); // new latch for new round
      } else {
        latch3.countDown();
        latch3.await();
        if (threadId == 0)
          latch2 = new CountDownLatch(NOF_THREADS); // new latch for new round
      }
      System.out.println("Round " + round + " thread " + threadId);
    }
  }
```

Oder mit zwei `CyclicBarrier` (weil dort das `countDown()` und `await()` zusammengehört (kein Kontexwechsel dazwischen möglich):

```
  private static CyclicBarrier latch1 = new CyclicBarrier(NOF_THREADS);
  private static CyclicBarrier latch2 = new CyclicBarrier(NOF_THREADS);

  private static void multiRounds(int threadId) throws InterruptedException, BrokenBarrierEx
    for (int round = 0; round < NOF_ROUNDS; round++) {
      if (round % 2 == 0) {
```

```
        latch1.await();
        if (threadId == 0)
          latch1.reset();
      } else {
        latch2.await();
        if (threadId == 0)
          latch2.reset();
      }
      System.out.println("Round " + round + " thread " + threadId);
    }
  }
```

Diese Lösung gefällt mir persönlich am besten.

3 Aufgabe 2: Dining Philosophers

3.1 Problemstellung

- Philosophen-Problem (gelb = essen; schwarz = denken, rot = hungern)

3.2 Betriebsmittelgraph

- Betriebsmittelgraph Beschreibung
  - Thread T wartet auf Lock von Ressource R: $T \rightarrow R$
  - Thread T besitzt Lock auf Ressource R: $R \rightarrow T$

Betriebsmittelgraph bei 5 Philosophen (`P[1-5]` = Philosophen; `G[1-5]` = Gabeln)

```
P1 → G1 → P2 → G2 → P3 → G3 → P4 → G4 → P5 → G5 → P1 → ... // Weiter am Anfang
```

Der Loop kann mit der topologischen Sortierung mit dem Tool `tsort` überprüft werden:

```
$ tsort
P1 G1
G1 P2
P2 G2
G2 P3
P3 G3
G3 P4
P4 G4
G4 P5
P5 G5
G5 P1

tsort: -: input contains a loop:
[...]
```

Es entsteht also ein Deadlock.

3.3 Analyse Korrekturvorschlag

Was ist an folgendem Lösungsvorschlag falsch?

```
table.acquireFork(leftForkNo);
while (!table.tryAcquireFork(rightForkNo)) {
  System.out.println("Philosophers " + getId() + " retries...");
  table.releaseFork(leftForkNo);
  sleep(500);
  table.acquireFork(leftForkNo);
}
```

Das heisst:

1. Nehme Gabel links
2. Rechte Gabel besetzt? (Wiederhole bis rechte Gabel in der Hand)
    a) Ja: Lege linke Gabel ab, warte und nehme die linke Gabel wieder auf
    b) Nein: Nehme auch die rechte Gabel

Das Problem Die Linke Gabel wird also in jedem Falle aufgenommen. Wie man in Teilaufgabe 1 sieht, ist die der Fall eines Deadlocks.

Lösung Es muss also einen Weg geben, dass nicht jeder die Gabel links aufnehmen kann.

3.4 Deadlock durh lineare Sperrordnung beheben

Wie gesagt, es muss einen Weg geben, dass nicht jeder die Gabel links von sich aufnehmen kann, damit alle bei der Aufnahme der zweiten Gabel blockiert sind (selbes gilt auch auf die andere Richtung, wenn die rechte Gabel zuerst genommen wird).

Folgende Lösung nimmt zuerst die Gabel mit der tiefsten Nummer:

```
if (leftForkNo < rightForkNo) {
  table.acquireFork(leftForkNo);
  table.acquireFork(rightForkNo);
} else {
  table.acquireFork(rightForkNo);
  table.acquireFork(leftForkNo);
}
```

4 Aufgabe 3: Upgradeable Read-Write Locks

Es soll ein Read Lock `upgradeableReadLock` implementiert werden, welcher später zu einem Write Lock konvertiert werden kann. Beispiel der Verwendung:

```
rwLock.upgradeableReadLock();
if (!contains(x) == 0) {
  rwLock.writeLock();
  add(x);
  rwLock.writeUnlock();
}
rwLock.upgradeableReadUnlock();
```

4.1 Warum ein spezieller Read Lock?

Wieso soll der Upgrade-Wunsch bereits beim Read Lock speziell mit upgradeableReadLock() bekannt gegeben werden?

Es gilt folgendes:

- Es kann *n* Read Locks auf die selbe Ressource gleichzeitig geben
- Es kann *n* Read Locks und 1 Upgradeable Read Lock auf die selbe Ressource gleichzeitig geben
- Es kann nur 1 Write Lock auf die selbe Ressource gleichzeitig geben

Zur Frage:

- Der Upgradeable Read Lock muss prüfen, ob es ein aktiver Read Lock gibt, falls nicht, kann er zum Write Lock upgraden.
- Da es gleichzeitig nicht mehrere Write Locks geben kann, aber mehrere Read Locks, muss man sicherstellen können, dass es pro Ressource nicht mehrere Upgradeable Read Locks gibt, welche zu einem potentiellen Write Lock werden können.

4.2 Implementation

Implementieren Sie eine solche Upgradeable Read-Write Lock Klasse (Fairness ist nicht hier zwingend). Sie können dazu das Gerüst und die Unit Tests aus der Vorlage verwenden. Intern können Sie einen Synchronisationsmechanismus Ihrer Wahl benutzen.

Meine Implementation sieht so aus:

```
package aufgabe3b;

public class UpgradeableReadWriteLock {
  private int readLocks = 0;
  private int upgradeableReadLocks = 0;
  private int writeLocks = 0;

  public synchronized void readLock() throws InterruptedException {
    while (writeLocks > 0)
      wait();
    ++readLocks;
```

```
  }

  public synchronized void readUnlock() {
    --readLocks;
    notifyAll();
  }

  public synchronized void upgradeableReadLock() throws InterruptedException {
    while (upgradeableReadLocks > 0 || writeLocks > 0)
      wait();
    ++upgradeableReadLocks;
  }

  public synchronized void upgradeableReadUnlock() {
    --upgradeableReadLocks;
    notifyAll();
  }

  public synchronized void writeLock() throws InterruptedException {
    while (readLocks > 0 || upgradeableReadLocks > 0 || writeLocks > 0)
      wait();
    ++writeLocks;
  }

  public synchronized void writeUnlock() {
    --writeLocks;
    notifyAll();
  }
}
```

Anmerkungen

- Mir ist erst jetzt richtig klar geworden, warum es das `while (foo) wait();` braucht, statt ein `if(foo) wait();`: Wird ein Thread aufgeweckt, muss er vielleicht nochmals eine Runde warten, weil ein Thread vor ihm an die Reihe kam.
- Release Methoden benachrichtigenn die Lock Methoden mittels `notifyAll()`.

4.3 Starvation verhindern

Realisieren Sie eine gewisse Fairness in der Synchronisationsprimitive: Writers sollen nicht kontinuierlich von neuen Readers überholt werden können (Starvation). (fakultativ, kompliziert)