

0. Hardware und Compiler:

Stalking the Lost Write: Memory Visibility in Concurrent Java

Jeff Berkowitz, New Relic

December 2013

(im gleichen Verzeichnis).

Schauen Sie sich die sog. „Executions“ der beiden Threads nochmal an. Erkennen Sie, dass es nur die zufällige Ablaufreihenfolge der Threads ist, die die verschiedenen Ergebnisse erzeugt? Das sind die berüchtigten „race conditions“...

Tun Sie mir den Gefallen und lassen Sie das Programm Mycounter in den se2_examples selber ein paar Mal ablaufen. Tritt die Race Condition auf? Wenn Sie den Lost Update noch nicht verstehen: Bitte einfach nochmal fragen!

Tipp: Wir haben ja gesehen, dass `i++` von zwei Threads gleichzeitig ausgeführt increments verliert (lost update). Wir haben eine Lösung gesehen in MyCounter: das Ganze in einer Methode verpacken und die Methode „synchronized“ machen. Also die „Ampellösung“ mit locks dahinter. Das ist teuer und macht unseren Gewinn durch mehr Threads kaputt.

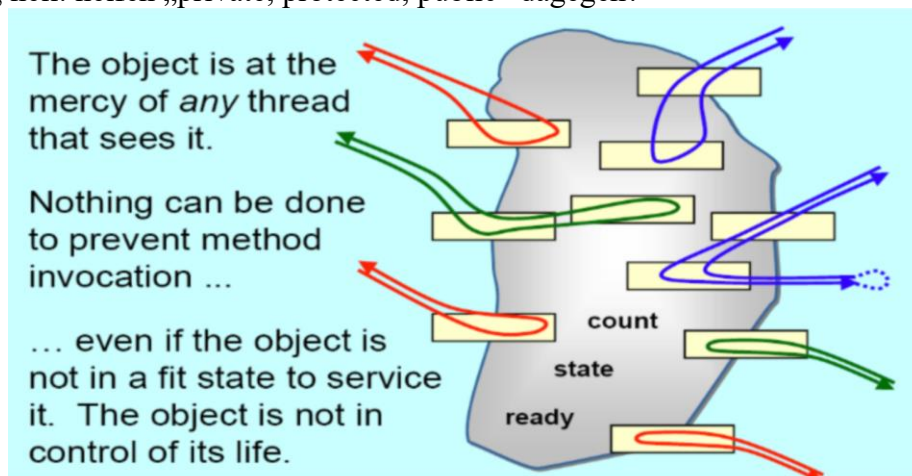
Hm, könnte man `i++` ATOMAR updaten ohne Locks??

Suchen Sie mal nach „AtomicInteger“ im concurrent package von Java!

1. Arbeitet Ihr Gehirn concurrent oder parallel (-:-)?

Parallel, da verschiedene Sinne wie Geruchs, Geschmack und Hörsinn gleichseitig verarbeitet werden.

2. Multi-threading hell: helfen „private, protected, public“ dagegen?



Ist Ihnen klar warum die OO-Kapselung durch „private“ Felder nicht hilft bei Multithreading?

Private bringt gar nichts, wenn die Threads über eine Methode an die Felder rankommen.

3. Muss der schreibende Zugriff von mehreren Threads auf folgende Datentypen synchronisiert werden?

- statische Variablen
- Felder/Attribute in Objekten
- Stackvariablen
- Heapvariablen (mit new() allokierte..)

Ja, bei allen außer den Stackvariablen, da hier jeder Thread einen eigenen Stack hat. Bei den anderen muss immer das Objekt mit einem Ampelsystem geschützt werden, wenn die Referenz auch noch andere Threads besitzen.

4. Machen Sie die Klasse threadsafe (sicher bei gleichzeitigem Aufruf). Gibt's Probleme? Schauen Sie ganz besonders genau auf die letzte Methode getList() hin.

```
class Foo {  
    private ArrayList aL = new ArrayList();  
    public int elements;  
    private String firstName;  
    private String lastName;  
  
    public Foo () {};  
  
    public void synchronized setfirstName(String fname) {  
        firstName = fname;  
        elements++;  
    }  
    public setlastName(String lname) {  
        lastName = lname;  
        elements++;  
    }  
    public synchronized ArrayList getList () {  
        return aL;  
    }  
}
```

setlastName Methode hat kein synchronized, damit darf jede Methode darauf zugreifen (Keine Ampel).

Variable elements ist public, jeder kann die Variable verändern.

Die synchronized ArrayList schützt nur die Referenz und nicht die Array Liste. Somit synchronized nicht nötig.

5. kill_thread(thread_id) wurde verboten: Wieso ist das Töten eines Threads problematisch? Wie geht es richtig?

Bei kill wird der Prozess nicht richtig beendet, sondern direkt gestoppt. So können die Daten dann verteilt rum liegen, wenn beim Übertragen der Prozess gestoppt wird.

Besser wenn sich der Thread selbst beenden kann.

Lieber .interrupt verwenden, damit sich der Thread selber beenden kann.

Oder eine Variable erstellen, die dann auf true gestellt wird, damit der Thread erkennt, dass er sich beenden muss und die Daten noch aufräumen kann.

6. Sie lassen eine Applikation auf einer Single-Core Maschine laufen und anschliessend auf einer Multi-Core Maschine. Wo läuft sie schneller?

Führer Single Core, heute kommt es darauf an, wie gut das Programm Multi-Core unterstützt. Z.B durch mehrere Threads die parallel im Programm laufen.

7. Was verursacht Non-determinismus bei multithreaded Programmen? (Sie wollen in einem Thread auf ein Konto 100 Euro einzahlen und in einem zweiten Thread den Kontostand verzehnfachen)

Die Reihenfolge der Thread. Ob zuerst auf das Konto eingezahlt wird, oder ob zuerst das Konto verzehnfacht wird.

8. Welches Problem könnte auftreten wenn ein Thread produce() und einer consume() aufruft? Wie sähen Lösungsmöglichkeiten aus?

```
1 public class MyQueue {
2
3     private Object store;
4     int flag = false; // empty
5
6     public void produce(Object in) {
7         while (flag == true) ; //full
8         store = in;
9         flag = true; //full
10    }
11    public Object consume() {
12        Object ret;
13        while (flag == false) ; //empty
14        ret = store;
15        flag = false; //empty
16        return ret;
17    }
18 }
```

Problem ist, das flag nur am Anfang auf den Ram zugreift und den Wert der flag abfragt und diesen dann in einen Cache setzt. Da sich dieser Cache nicht ändert, wäre die loop in produce() unendlich. In consume() ändert sich aber der Zustand von flag. Deswegen müssen wir flag auf volatile setzen, damit es immer den neusten Stand vom Ram abfragt.

9. Fun with Threads:

1. Was passiert mit dem Programm?
2. Was kann auf der Konsole stehen?

```
public class C1 {
public synchronized void doX (C2 c2) {
    c2.doX();
}
public synchronized void doY () {
    System.out.println("Doing Y");
}

public static void main (String[] args) {
    C1 c1 = new C1();
    C2 c2 = new C2();
    Thread t1 = new Thread(() -> { while (true) c1.doX(c2);});
    Thread t2 = new Thread(() -> { while (true) c2.doY(c1);});
    t1.start();
```

```
t2.start();  
}  
};
```

```
public class C2 {
```

```
public synchronized void doX () {  
    System.out.println("Doing X");  
}
```

```
public synchronized void doY ( C1 c1) {  
    c1.doY();  
}
```

1. Beide Threads halten ihre Ressourcen fest und wollen jeweils von anderem die Resource.
Verkleben sich und wir haben einen Deadlock -> Programm bleibt stehen.
2. Doing X