Session: Concurrency and Parallelism in Java

Sara Tietze (st093), Merve Özdemir(mo064), Pia Schilling (ps149)

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“…

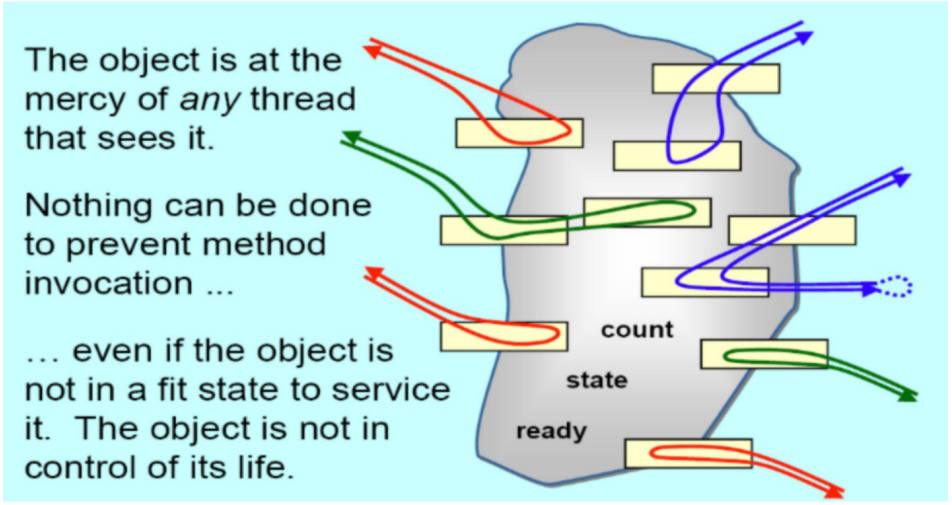
Tun Sie mir den Gefallen und lassen Sie das Programm Mycounter in den se2_examples selber ein paar Mal ablaufen. Tritt die Race Condition auf? Wenn Sie den Lost Update noch nicht verstehen: Bitte einfach nochmal fragen!
Tipp: Wir haben ja gesehen, dass i++ von zwei Threads gleichzeitig ausgeführt increments verliert (lost update). Wir haben eine Lösung gesehen in MyCounter: das Ganze in einer Methode verpacken und die Methode „synchronized“ machen. Also die „Ampell sung“ mit locks dahinter. Das ist teuer und macht unseren Gewinn durch mehr Threads kaputt.

Hm, könnte man i++ ATOMAR updaten ohne Locks?? Ja
Suchen Sie mal nach „AtomicInteger“ im concurrent package von Java!

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

In dem menschlichen Hirn können/müssen mehrere Prozesse gleichzeitig ablaufen. Daher arbeitet unser Gehirn parallel.

2. Multi-threading hell: helfen „private, protected, public“ dagegen?
Ist Ihnen klar warum die OO-Kapselung durch „private“ Felder nicht hilft bei Multithreading?



Ja uns ist klar, dass das private Deklarieren von Feldern keinerlei Schutz vor Threads bietet .Die Felder sind durch public Methoden trotzdem für die Threads sichtbar.

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

- **statische Variablen** → ja da Klassenvariablen nur einmal pro Klasse vorhanden sind. Das heißt jeder Thread, der mit dieser Klasse arbeitet wird auch diese Variable evlt verändern

- **Felder/Attribute in Objekten** → ja, die Objekte werden (mit ihren Feldern) im Heap gespeichert. Das heißt, dass jeder Thread Stack zwar seine eigene Referenz auf das Objekt hat. Jedoch zeigt die Referenz auf dasselbe Objekt und damit auf die selben Felder.

- Stackvariablen → Nein, da jeder Thread sein eigener Stack besitzt und sie sich so "nicht in die Quere kommen"

- Heapvariablen (mit new() allokierte..) → kommt drauf an wo das "new" allokiert wird. Wenn es in einer Methode allokiert wird, dann kann es unsynchronisiert bleiben. Wenn es allerdings in einem Field erstellt wird, kann es sein, dass es auf die selbe Variable im Stack referenziert.

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 void synchronized setlastName(String lname) {
		lastName = lname;
		elements++;
	}

	public synchronized ArrayList getList () {
		return aL;
	}
}
```

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

Weil Deadlocks öfter vorkommen können, stattdessen kann man den Thread mit sleep(), zum schlafen legen oder mit yield() das zeitlich schedulen. Außerdem kann es so zu ungültigen Objektzuständen kommen, da es sein kann, dass der Thread gerade Objekte modifiziert, wenn er gekillt wird.

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

Das ist davon abhängig, wie die Applikation programmiert ist. Wurde die Applikation nicht für Multithreading konzipiert, wird sie auf einer Multi-Core Maschine auch nicht schneller laufen, da dann trotzdem nur ein Core verwendet wird.

Eine Multicore Maschine hat vielleicht eine schnellere Reaktionszeit und kann komplexere Aufgaben bearbeiten, aber es muss nicht sein, dass Applikationen schneller laufen. Es kommt noch auf weitere Aspekte an.

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)

Das Verhalten der Threads ist nie vorhersehbar. Es kann sein, dass bei Programmablauf 1 Thread A als erstes fertig ist und beim nächsten Programmablauf Thread B als erstes fertig ist. In dem gegebenen Beispiel würden dementsprechend immer verschiedene Ergebnisse rauskommen, je nachdem welcher Thread schneller ist.

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

```
1 public class MyQueue {
2
3     private Object store;
4     private boolean flag = false; // empty          → why int?!

5     public void produce(Object in) {
6           while (flag == true) ; //full
7           store = in;
            flag = true; //full
9     }

10     public Object consume() {
            Object ret;
11          while (flag == false) ; //empty
12          ret = store;
13          flag = false; //empty
14          return ret;
15    }
16 }
```

Der producer sollte nicht versuchen neue Daten einzufügen wenn er schon voll ist und der consumer sollte nicht versuchen Daten zu entfernen wenn der Buffer schon leer ist. Der consumer sollte benachrichtigt werden (notify()), wenn das Objekt nicht mehr leer ist. Bis dahin sollte der consumer darauf warten (wait()). Außerdem sollte synchronized verwendet werden, da es sonst evtl zu einem gleichzeitigen Zugriff auf das Objekt kommen kann. Es sollte zudem darauf geachtet werden, dass es nicht zu einem Deadlock kommt, indem der consumer auf den producer wartet und umgekehrt.

9. Fun with Threads:

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

```
public class C2 {
	public synchronized void doY ( C1 c1) {
		c1.doY();
	}

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

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();
	}
	};
```

1. Es wird eine Endlosschleife generiert.

2. Auf der Konsole könnte stehen: Doing X (doY kommt nicht zum Zug, weil C2 von der Methode doX durch das Schlüsselwort synchronized blockiert wird. Entfernt man das Schlüsselwort, kommt auch Thread 2 zum Zug.