HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Blatt 2

Betriebssysteme

Luis Staudt

c) maximale Anzahl an Threads

Auf zwei Systemen getestet: Beide liefen über eine halbe Stunde ohne Probleme. Ich vermute, dass es keine Begrenzung für die Anzahl der Threads gibt. Es könnte eine Limitierung durch die Systemressourcen geben.

d) Programmabläufe mit unterschiedlichen Ergebnissen

Endergebnis: 0 und 1

In diesem Fall führt zuerst Thread X die ersten drei Zeilen seiner `tausche`-Methode auf Objekt a0 aus. Dadurch wird der Wert von a0 zu 1 und der Wert von a1 zu 0. Anschließend führt Thread Y die komplette `tausche`-Methode auf Objekt a1 aus. Hierbei wird der Wert von a1 wieder auf 1 gesetzt und der Wert von a0 auf 0. Am Ende haben wir also die ursprünglichen Werte.

Zeilennr. Thread		Objekt a0		Objekt a1	
X in a0	Y in a1	wert	h	wert	h
1	-	0	0	1	-
2	-	1	0	1	-
3	-	1	0	0	-
-	1	1	0	0	0
-	2	1	0	1	0
-	3	0	0	1	0

Tabelle 1: Thread-Ausführungsreihenfolge für Endergebnis: 0 und 1

Endergebnis: 1 und 1

Bei dieser Ausführungsreihenfolge führt zuerst Thread Y Zeile 1 seiner `tausche`-Methode aus und speichert den Wert 1 von a1 in seiner lokalen Variable h. Dann führt Thread X alle drei Zeilen seiner `tausche`-Methode aus, wodurch a0 den Wert 1 erhält und a1 den Wert 0. Anschließend führt Thread Y die restlichen Zeilen 2 und 3 aus, wodurch a1 wieder Wert 1 erhält und a0 ebenfalls Wert 1 (aus der in Zeile 1 gespeicherten Variable h).

Endergebnis: 1 und 0

Hier führt Thread X zuerst Zeile 1 seiner `tausche`-Methode aus und speichert den Wert 0 von a0 in seiner lokalen Variable h. Dann führt Thread Y alle drei Zeilen aus, wodurch a1 den Wert 0 erhält und a0 den Wert 1. Anschließend führt Thread X die restlichen Zeilen 2 und 3 aus, bestätigt den Wert 1 für a0 und setzt a1 auf 0.

Zeilennr. Thread		Objekt a0		Objekt a1	
X in a0	Y in a1	wert	h	wert	h
-	1	0	-	1	1
1	-	0	0	1	1
2	-	1	0	1	1
3	-	1	0	0	1
-	2	1	0	1	1
-	3	1	0	1	1

Tabelle 2: Thread-Ausführungsreihenfolge für Endergebnis: 1 und 1

Zeilennr. Thread		Objekt a0		Objekt a1	
X in a0	Y in a1	wert	h	wert	h
1	-	0	0	1	-
-	1	0	0	1	1
-	2	1	0	1	1
-	3	1	0	0	1
2	-	1	0	0	-
3	-	1	0	0	-

Tabelle 3: Thread-Ausführungsreihenfolge für Endergebnis: 1 und 0

Endergebnis: 0 und 0

Bei dieser Ausführungsreihenfolge werden verschiedene Zeilen der `tausche`-Methode durch die Threads ausgeführt, wobei am Ende beide Objekte den Wert 0 haben. Thread Y führt seine Zeilen in einer bestimmten Reihenfolge aus, setzt a0 auf 1 und a1 auf 0. Anschließend führt Thread X Zeilen 2 und 3 aus, wobei hier besonders der Unterschied zum vorherigen Fall ist, dass a0 bereits vor Zeile 2 den Wert 0 besitzt und somit der Wert von a0 bei 0 bleibt.

Zeilennr. Thread		Objekt a0		Objekt a1	
X in a0	Y in a1	wert	h	wert	h
1	-	0	0	1	-
-	1	0	0	1	1
-	2	0	0	0	1
-	3	1	0	0	1
2	-	0	0	0	-
3	-	0	0	0	-

Tabelle 4: Thread-Ausführungsreihenfolge für Endergebnis: 0 und 0

Änderung des Verhaltens mit privater Synchronisation

- Das Verhalten des Programms gegenüber I) ändert sich **nicht**.
- Begründung: Jedes A-Objekt hat sein eigenes privates Lock-Objekt `o`.

- Wenn Thread X auf `a0.tausche(a1)` zugreift, synchronisiert es auf das private Lock von `a0`.
- Wenn Thread Y auf `a1.tausche(a0)` zugreift, synchronisiert es auf das private Lock von `a1`.
- Da es sich um verschiedene Lock-Objekte handelt, können beide Threads immer noch parallel ausgeführt werden.
- Die gleichen vier Ergebnisse wie in Teil I) sind weiterhin möglich.

Die private Synchronisation bewirkt lediglich, dass die `tausche`-Methode atomar in Bezug auf dasselbe Objekt ausgeführt wird. Da die beiden Threads jedoch auf unterschiedlichen Objekten operieren und somit unterschiedliche Locks verwenden, bleibt die Race-Condition zwischen den Threads bestehen.

Änderung des Verhaltens mit statischer Synchronisation

- Mit einem statischen Lock-Objekt ändert sich das Verhalten grundlegend.
- Da alle A-Objekte dasselbe Lock verwenden, kann zu jedem Zeitpunkt nur ein Thread in den synchronisierten Block eintreten.
- Dies bedeutet:
  - Die Methode `tausche` wird atomar ausgeführt
  - Es sind keine Verzahnungen von Threads mehr möglich
  - Nur zwei Ausführungsreihenfolgen sind noch möglich, die aber zum gleichen Endergebnis führen
- Ergebnis in beiden Fällen: `a0.wert = 0, a1.wert = 1`
  - Fall 1: Thread X führt vollständig aus, dann Thread Y
  - Fall 2: Thread Y führt vollständig aus, dann Thread X
- In beiden Fällen tauschen sich die Werte erst komplett und dann wieder zurück.

Durch die Verwendung eines statischen Lock-Objekts wird ein globaler Synchronisationspunkt für alle Instanzen der Klasse A geschaffen. Dies verhindert jegliche verzahnte Ausführung der `tausche`-Methode und garantiert, dass immer nur ein Thread zu einem Zeitpunkt innerhalb des synchronisierten Blocks ist. Dadurch reduzieren sich die möglichen Endzustände auf nur einen Fall.

Bewertung der unterschiedlichen Synchronisationsmethoden

- **Methode III (statisches Lock)** ist besser geeignet, weil:
  - Sie verhindert alle möglichen Race Conditions
  - Sie ist einfacher zu verstehen
  - Sie vermeidet Deadlocks
- **Methode IV (verschachtelte Locks)** hat ein kritisches Problem:
  - Sie kann zu Deadlocks führen, wenn zwei Threads die Methode gleichzeitig mit vertauschten Argumenten aufrufen
  - Beispiel für Deadlock:
    - Thread X ruft `a0.tausche(a1)` auf und erhält das Lock für `a0`
    - Thread Y ruft `a1.tausche(a0)` auf und erhält das Lock für `a1`
    - Thread X wartet auf das Lock für `a1` (das Y hält)
    - Thread Y wartet auf das Lock für `a0` (das X hält)
    - Beide Threads blockieren sich gegenseitig - klassischer Deadlock

Die Methode mit statischem Lock bietet den Vorteil der Einfachheit und Sicherheit, indem sie globale Synchronisation gewährleistet. Die verschachtelte Synchronisation in Methode IV führt zu einem klassischen Deadlock-Problem durch zyklische Abhängigkeiten, was in produktiven Umgebungen kritisch wäre und vermieden werden sollte.