ParaProg Übung W04, Testat 02

Parallele Prgrammierung

Emanuel Duss

2014-04-15 16:26

Inhaltsverzeichnis

1 Ziele

- Java und .NET Memory Model vertiefen.
- Memory Fences, volatile und atomare Operationen anwenden.
- Eigene Lock-freie Thread-sichere Datenstruktur implementieren.

2 Aufgabe 1: Peterson's Algorithmus

- Peterson's Algorithmus: Gegenseitigen Ausschluss von Threads mit lediglich Lese- und Schreibzugriffen.

2.1 Implementierung

```
// Initialisierung
boolean state0 = false;
boolean state1 = false;
int turn = 0;

// Thread 0
state0 = true;
turn = 1;
while (turn == 1 && state1);
  // critical section
state0 = false;

// Thread 1
state1 = true;
turn = 0;
while (turn == 0 && state0);
  // critical section
state1 = false;
```

2.2 Effekte des Speichermodells

- Die Threads behalten die Variabeln zur Laufzeit in einem Cache, damit der Zugriff auf diese schneller ist.
- Es ist daher aber nicht sichergestellt, dass die Variablen `turn` und `state[01]` beim Auslesen auch wirklich den Wert beinhalten, welcher ihnen aktuell zugewiesen ist.
- Anweisungen, welche nicht atomar sind, können von der Laufzeitumgebung (JVM unter Java) in geänderter Reihenfolge ausgeführt werden, falls dabei die semantik nicht ändert.

2.3 Implementierung in Java

Die Variabeln müssen als `volatile` markiert sein, damit garantiert ist, dass alle Threads jederzeit den wirklich zugewiesenen Wert auslesen. Somit werden die Variabeln bei einer Änderung wirklich ins Memory geschrieben und vom Memory gelesen, anstatt vom Cache.

```
private volatile boolean state0 = false;
private volatile boolean state1 = false;
private volatile int turn = 0;
```

2.4 Implementierung in .NET C#

Unter .NET können Anweisungen trotz des `volatile` Keywords weiterhin in beliebiger Reihenfolge ausgeführt werden. Um wirklich der aktuelle Wert vom Memory zu lesen, muss man dies explizit

mit `Thread.MemoryBarrier();` mitteilen:

```
public void Thread0Lock(){
  state0 = true;
  turn = 1;
  System.Threading.Thread.MemoryBarrier();
  while (turn == 1 && state 1);
}
```

3 Aufgabe 2: Atomare Operationen

- Gemeinsames Objekt `BankAccount`
- Ziel: Implementierung ohne Locks mittels Java Atomic Klassen anstatt der Monitor Synchronisation.
- Ausführungsdauer mit Monitor Synchronisation: 2.1 Sekunden

Es wird ein `AtomicInteger` verwendet. Die Methode `withdraw(int amount)` ist am interessantesten. Wichtig ist, dass man den Wert von `balance` nur einmal liest und in eine Variable speichert, denn dann kann man sicher sein, dass man immer den selben Wert verwendet, auch wenn währen der Laufzeit der Inhalt der Variable erneut geändert wird.

```
private AtomicInteger balance = new AtomicInteger(0);

  public void deposit(int amount) {
    balance.getAndAdd(amount);
  }

  public boolean withdraw(int amount) {
    int oldValue;
    int newValue;
    do {
      oldValue =  balance.get();
      if (amount <= oldValue)
        newValue = oldValue - amount;
      else
        return false;
    } while (!balance.compareAndSet(oldValue, newValue));
    return true;
  }

  public int getBalance() {
    return balance.get();
  }
```

4 Aufgabe 3: Lock-freier Stack

- Ziel: Lock-freien und thread-sicheren Stack nach Treiber in Java implementieren.

Implementierung von `push` und `pop`:

```
public void push(T value) {
  Node<T> newNode = new Node<>(value);
  Node<T> current;
  do {
    current = top.get();
    newNode.setNext(current);
  } while (!top.compareAndSet(current, newNode));
}

public T pop() {
  Node<T> current;
  Node<T> newtop;
  do {
    current = top.get();
    if (current == null)
      return null;
    newtop = current.getNext();
  } while (!top.compareAndSet(current, newtop));
  return current.getValue();
}
```

Output:

- aufgabe3.SynchronizedStack: 22701ms
- aufgabe3.LockFreeStack: 17215ms

Tipp von Herr Bläser: Eine höhere Geschwindigkeit kann man erreichen, indem man die Verzahnung mittels `Thread.yield()` erhöht [1].

[1] Implementierung vgl. Sourcecode