Karlsruher Institut für Technologie

KASTEL / IPD

Prof. Dr. Sebastian Erdweg erdweg@kit.edu
Prof. Dr. Ralf H. Reussner reussner@kit.edu

Programmierparadigmen – WS 2025/26

https://dsis.kastel.kit.edu/1181_1197.php

Blatt 2: Java Parallelprogrammierung Besprechung: 09.01.2026

1 Amdahlsches Gesetz

Ein Programm mit mehreren Threads arbeitet auf einem Puffer. Ein Teil der Threads liest Daten aus dem Puffer (Leser), der andere Teil schreibt Daten in den Puffer hinein (Schreiber). Eine beliebige Anzahl lesender Threads kann gleichzeitig auf dem Puffer arbeiten. Wenn ein Schreiber auf den Puffer zugreift, kann kein anderer Thread (Leser oder Schreiber) auf dem Puffer aktiv sein.

Gegeben ist ein Thread Pool, wobei jeder Leser und Schreiber durch einen Thread repräsentiert wird. Dabei sind 90% der Threads Leser und 10% Schreiber. Jeder Leser benötigt 2 Sekunden und jeder Schreiber 3 Sekunden zur Ausführung. Das Programm ist beendet, wenn alle Threads im Pool beendet sind.

Aufgabe: Unter Berücksichtigung des Amdahlschen Gesetzes, wo liegt die obere Grenze der Beschleunigung (Speedup) der oben beschriebenen Implementierung auf einem 4-Kern-Prozessor?

2 Threading-Grundlagen: Producer-Consumer

Ein häufig verwendetes Muster in der Parallelverarbeitung zur Verteilung von Arbeit auf ein oder mehrere Arbeiter-Threads ist das Producer-Consumer-Muster. Hierbei werden in eine Datenstruktur Elemente von Erzeuger-Threads (Producer) eingefügt und von Verarbeitungs-Threads (Consumer) entfernt und verwendet.

Aufgabe: In dieser Aufgabe sollen Sie für ein Producer-Consumer-Szenario mit Integern die Basis-Funktionalität und das Threading implementieren. Die benötigten Source-Code-Dateien, in denen Sie arbeiten werden, finden Sie im ILIAS-Kurs.

1. Ergänzen Sie die Methoden `produce()` und `int consume()` in der bereitgestellten `Buffer`-Klasse in der `Buffer.java`-Quelldatei. In `produce()` soll der erzeugte Integer (`element`) der Warteschlange `queue` thread-sicher hinzugefügt werden. Falls dieser Vorgang die gegebene Maximal-Kapazität (`capacity`) von `queue` überschreiten würde, soll der Thread warten bis wieder Kapazitäten frei sind. In der Methode `int consume()` soll ein Element aus der Queue `queue` Thread-sicher entfernt werden, wenn mindestens ein Element vorhanden ist. Ansonsten soll der Thread warten bis Elemente hinzugefügt wurden.
   Sie können für das Warten auf Kapazitäten oder auf Elemente die Methoden `wait()` und `notifyAll()` verwenden.
   Tipp: Falls Sie die Ergebnisse auf der Konsole einfacher betrachten möchten, können Sie in die `produce` und `consume` Methoden Wartezeiten einfügen, damit die Ausgaben seltener erfolgen (z.B. durch Aufrufe von `Thread.sleep(int milliseconds)`).

2. Als nächstes sollen Sie zwei Producer-Threads und einen Consumer-Thread erstellen und starten. Modifizieren Sie hierfür die unten angegebene `main`-Methode der `ProducerConsumer.java`-Quelldatei. Alle Threads sollen die gleiche Instanz der `Buffer`-Klasse verwenden. Jeder Producer- bzw. Consumer-Thread soll die Methode `produce()` bzw. `consume()` der `Buffer`-Klasse solange aufrufen, bis der Wert der Kontroll-Variable `running` der `Buffer`-Klasse auf `false` gesetzt wird. Sie können den Wert durch die getter-Methode `isRunning()` abrufen. Bei der Implementierung des Threadings dürfen Sie alle in der Vorlesung vorgestellten Ansätze verwenden. Achten Sie darauf, eventuell wartende Threads zu beenden. Ergänzen Sie die folgende `main`-Methode und erstellen Sie bei Bedarf bis zu zwei weiteren Klassen:

```
public static void main(String[] args){
    Buffer buffer = new Buffer();

    //Erstellen und starten Sie die Threads hier.

    try {
        Thread.sleep(10000);
    } catch (InterruptedException e1) {
        buffer.stopRunning();
    } finally {
        buffer.stopRunning();
    }

    // Beenden Sie ihre Threads hier und warten Sie auf Vollendung

}
```

3. Warum ist es in diesem Producer-Consumer Schema besser `notifyAll()` und nicht `notify()` zu verwenden?

3 Synchronisation: Barrieren

Implementieren Sie eine einfache zyklische Barriere. Die Barriere bekommt als einzigen Konstruktorparameter die Anzahl an Threads, welche `await` aufrufen müssen, damit die Barriere alle Threads freigibt. Nach dem Freigeben soll die Barriere erneut benutzbar sein. Die Methode `freeAll` löst eine manuelle Freigabe aus. Alle Threads, welche vor dem Zeitpunkt des `freeAll`-Aufrufs `await` aufgerufen haben, werden freigegeben. Weitere Threads, welche nach dem `freeAll`-Aufruf kommen, müssen wieder warten. Implementieren Sie folgendes Interface:

```
public interface IBarrier {
  void await() throws InterruptedException;
  void freeAll();
}
```

Testen Sie Ihre Barriere.

4 Speicherkonsistenz: Happens-before-Beziehung

Die Happens-before-Beziehung zwischen zwei Statements in Java garantiert, dass ein Statement den geschriebenen Wert des anderen Statements sieht. Fehlende Happens-before-Beziehungen können im Multi-Threading schwer ersichtliche Auswirkungen auf die Programm-Korrektheit haben.

In folgendem Codeausschnitt ist ein Programm zu sehen, das ein Ping-Pong Spiel zwischen zwei Threads dargestellt. Hierbei wird eine `ping`-Variable von dem PPingThread `pingThread` auf `true` gesetzt und eine entsprechende Ausgabe auf die Konsole erstellt. Der PPongThread `pongThread` setzt die `ping`-Variable auf `false` und erstellt ebenfalls eine Ausgabe auf die Konsole.

```
public class HappensBefore {
    public static boolean ping = false;
    public static final int maxRuns = 100;

    public static void main(String[] args) {
        Thread pingThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(ping) {}

                ping = true;
                System.out.println("Ping␣-␣Round␣" + i);
            }
        });
        Thread pongThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(!ping) {}

                ping = false;
                System.out.println("Pong␣-␣Round␣" + i);
            }
        });

        pingThread.start();
        pongThread.start();
    }
}
```

Aufgabe: Bei mehrfacher Ausführung des Programmes terminiert es in einigen Fällen nicht. Wieso kommt es zu diesem Verhalten? Verwenden Sie für Ihre Erklärung als Argumentationsgrundlage Happens-before-Beziehungen. Wie lassen sich die Probleme beheben?

5 Fortgeschrittene Parallelisierungsprinzipien: Max Of Max

Gegeben sei die Klasse MaxOfMax welche die `calculateMax` bereitstellen soll. Die `calculateMax` Methode soll den größten Wert einer Sequenz von `Integer`-Zahlen parallel berechnen. In einem Vorverarbeitungsschritt wird die Sequenz in disjunkte Blöcke aufgeteilt und über den Parameter `blocksOfNumbers` der Methode übergeben. Für jeden Block soll der größte Wert berechnet und folgend der größte Wert über die Ergebnisse aller Blöcke bestimmt werden. Der größte Wert einer Folge von `Integer`-Zahlen wird mittels der ebenfalls bereitgestellten Methode `findMax` berechnet. Gehen Sie in den folgenden Aufgaben davon aus, dass alle Datenstrukturen und `Integer`-Objekte in dem Parameter `blocksOfNumbers` instanziiert sind, weswegen Sie das Auftreten und Behandeln von `null`-Werten nicht berücksichtigen müssen.

Aufgabe: In dieser Aufgabe sollen Sie Teile der Klasse MaxOfMax implementieren. Verwenden Sie hierfür die Source-Code Datei `MaxOfMax.java`, welche Ihnen im ILIAS-Kurs zur Verfügung gestellt wird.

1. Vervollständigen Sie die Implementierung der `calculateMax`-Methode in der Datei `MaxOfMax.java` unter Verwendung von `ExecutorService` und `Futures`. Das Auftreten von Exceptions muss nicht beachtet werden.

2. Ergänzen Sie die Methode `findMaxStream` in der Datei `MaxOfMax.java`, welche als Ersatz für die Methode `findMax` dienen soll. Verwenden Sie hierfür parallele Java Streams und die Stream-Operation `max`.
   Hinweis: Beachten Sie den Rückgabe-Typ der Stream-Operation `max` des von Ihnen verwendeten Stream-Typs. Der Rückgabe-Typ von `Stream::max` ist beispielsweise ein `Optional<T>`[1].

[1] https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html