Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion

Prof. Dr. Michael Hanus, Niels Bunkenburg, M.Sc. Finn Teegen

Institut für Informatik

2. Übung zur Vorlesung „Fortgeschrittene Programmierung“

Nebenläufigkeit II
Abgabe am Montag, 16. November 2020 - 10:15

---

2. Übung Fortgeschrittene Programmierung

Nebenläufigkeit 2

Bitte melden Sie sich im Mattermost-Team für FortProg an! Die Organisation der Veranstaltung findet primär über Mattermost statt.

Für die Teilnahme an den Übungen empfehlen wir, den Zoom Client zu installieren bzw. zu aktualisieren, da der Browserclient bzw. ältere Versionen des Desktopclients nicht alle Features unterstützen.

Präsenzaufgabe 1 - Konfligierende Philosophen

Betrachten Sie die folgende, **fehlerhafte** Implementierung der dinierenden Philosophen mit Deadlockvermeidung durch “Ziehen mit Zurücklegen”.

Diese Implementierung beinhaltet drei Fehler, die zu einem fehlerhaften Verhalten führen können. Geben Sie für jeden dieser Fehler

- die fehlerhafte Codestelle,
- eine Erklärung, warum diese Stelle fehlerhaft ist,
- sowie eine fehlerbereinigte Implementierung an.

Gehen Sie davon aus, dass die Sticks den Philosophen korrekt übergeben werden.

```
1  public class Stick {
2
3          private boolean isUsed;
4          private int num;
5
6          public Stick(int num) {
7                  isUsed = false;
8                  this.num = num;
9          }
10
11         public synchronized void put() {
12                 isUsed = false;
13         }
14
15         public synchronized void get() {
16                 if (isUsed) {
17                         try {
```

```
                                wait();
                        } catch (InterruptedException e) {}
                }
                isUsed = true;
        }

        public synchronized boolean lookFor() {
                return !isUsed;
        }
}
```

```
public class Philosopher implements Runnable {

        private Stick left, right;
        private int num;

        public Philosopher(int num, Stick left, Stick right) {
                this.num = num;
                this.left = left;
                this.right = right;
        }

        public void run() {
                while (true) {
                        try {
                                System.out.println("Thinking.");

                                // Ziehen mit Zurücklegen
                                left.get();
                                if (!right.lookFor()) {
                                        synchronized (right) {
                                                right.wait();
                                        }
                                }
                                right.get();
                                System.out.println("Eating.");
                                left.put();
                                right.put();
                        } catch(InterruptedException e) {}
                }
        }
}
```

Aufgabe 2 - Erweiterung der Klasse Account 1 Punkt

In dieser Aufgabe sollen Sie mit der folgenden Vorlage der `Account`-Klasse arbeiten (der Code lässt sich per Klick auf **Details** anzeigen).

1. Erweiteren Sie die Klasse `Account` um eine Methode `transfer`, welche Geld von einem auf ein anderes Konto überweist. Definieren Sie auch eine Methode `safeTransfer`, welche nur überweist, falls genügend Geld vorhanden ist.
2. Das Paket `java.util.concurrent` enthält auch eine Klasse `Semaphore` (JavaDoc) , welche die Methoden `acquireUninterruptibly` und `release` für die Operationen P und V zur Verfügung stellt. Ersetzen

Sie die `synchronized`-Methoden durch die Verwendung einer Semaphore zum Schützen der kritischen Bereiche.

3. Vergleichen Sie beide Implementierungen bezüglich ihres Verhaltens. Gibt es Unterschiede bei konkreten Verwendungen der Klasse `Account`? Wenn ja, erklären Sie die Ursache des Unterschieds.
4. Sowohl die Implementierung der Vorlesung (erweitert um Aufgabenteil 1) als auch die Implementierung mit Semaphoren haben Deadlocks. Überlegen Sie sich eine Variante, welche frei von Deadlocks ist.

Aufgabe 3 - Beschränkter Puffer

1 Punkt

Implementieren Sie eine Klasse `BufferN<E>`, die einen in der Größe beschränkten Puffer mit Hilfe eines internen Arrays realisiert.

1. Die Kapazität n des Puffers soll als Argument im Konstruktor angegeben werden. Für eine ungültige Kapazität ($n \leq 0$) soll eine `IllegalArgumentException` geworfen werden.
2. Die Klasse `BufferN<E>` soll die folgenden Methoden umfassen:
   - `public synchronized E take()` liest den jeweils ältesten Eintrag aus und entfernt diesen aus dem Puffer. Ist der Puffer leer, so suspendiert die Methode.
   - `public synchronized void put(E elem)` fügt einen Wert in den Puffer ein. Ist der Puffer voll, so suspendiert die Methode, bis der Wert eingefügt werden kann.
   - `public synchronized boolean isEmpty()` gibt zurück, ob der Puffer leer ist.

Geben Sie eine sichere Implementierung an, in der gegebenenfalls auch zu viel synchronisiert wird.

Aufgabe 4 - Erweiterung des einelementigen Puffers

1 Punkt

1. Erweitern Sie die verbesserte Variante der Klasse `Buffer1` (mit zwei Synchronisationsobjekten) aus der Vorlesung (der Code lässt sich per Klick auf **Details** anzeigen)

   um folgende Methoden:

   - `boolean tryPut(E elem)` soll `elem` in den Puffer eintragen und `true` zurückgeben, falls der Puffer leer ist. Falls nicht, soll die Methode im Gegensatz zu `put` nicht suspendieren sondern `false` zurückgeben.
   - `E read()` soll den Pufferinhalt auslesen ohne den Puffer zu leeren. Ist der Puffer leer, so suspendiert die Methode.
   - `void overwrite(E elem)` soll den Pufferinhalt mit `elem` überschreiben ohne zu suspendieren, unabhängig davon ob der Puffer gerade voll ist oder nicht.
   - Geben Sie für die Methode `take` eine Variante `take (long timeout)` an, die nur eine übergebene Zeit (in msec) lang suspendiert. Wenn die Zeit `timeout` abgelaufen ist, ohne dass ein Wert gelesen werden konnte, soll eine `java.util.concurrent.TimeoutException` geworfen werden. Achten Sie darauf, dass Sie nicht übermäßig lange warten!