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

1. Übung zur Vorlesung „Fortgeschrittene Programmierung“

Nebenläufigkeit I

Abgabe am Montag, 09. November 2020 - 10:00

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, da die Browserversion nicht alle Features zuverlässig unterstützt.

Präsenzaufgabe 1 - Counter

1. Definieren Sie eine Klasse `Counter`, die die Klasse `Thread` beerbt und einen sich ständig inkrementierenden Zähler (beginnend bei 0) realisiert. Die Geschwindigkeit, mit der der Zähler hochgezählt wird, soll durch einen Zeitparameter (in msec) des Konstruktors festgelegt werden. Diese Zeit soll der Zähler warten, bis er das nächste Mal hochzählt. Außerdem soll der Zähler bei seiner Konstruktion noch einen Namen erhalten. Bei jedem Hochzählen soll der Zähler seinen Namen und den aktuellen Wert auf die Standardausgabe schreiben.
2. Schreiben Sie ein Programm mit dem es möglich ist, beliebig viele Zähler nebenläufig zu starten. Dazu soll beim Programmstart für jeden gestarteten Zähler die Geschwindigkeit angegeben werden. Drei unterschiedlich schnelle Zähler sollen beispielsweise durch den folgenden Befehl gestartet werden.

   ```
   java CounterMain 300 500 1000
   ```

3. Stellen Sie mittels `synchronized`-Blöcken sicher, dass das Schreiben auf die Standardausgabe atomar stattfindet.
4. Verändern Sie die Klasse `Counter` so, dass sie die Schnittstelle `Runnable` direkt implementiert (keine Vererbung) und passen Sie das Programm an. Was spricht für diese Lösung, was dagegen?

Aufgabe 2 - Synchronisierte Philosophen — 1 Punkt

In der Vorlesung haben Sie das Problem der dinierenden Philosophen kennengelernt. Sie sollen nun eine erste Version der dinierenden Philosophen in Java implementieren.

1. Implementieren Sie die dinierenden Philosophen in Java. Realisieren Sie den gegenseitigen Ausschluss dabei nur unter Verwendung von `synchronized`-Methoden bzw. -Blöcken und verzichten Sie insbesondere auf `wait()` und `notify()`.

   Ihre Implementierung sollte sich an dem folgenden Gerüst orientieren.

   - Die Philosophenklasse ist ein Thread oder implementiert das Interface `Runnable`.
   - Philosophen besitzen eine Nummer und zwei Objekte, die den linken und rechten Stick repräsentieren. Diese Attribute werden dem Konstruktor übergeben.

- Philosophen haben die Methoden `snooze`, welche den Philosophen eine zufällige Dauer verweilen lässt, und `run`, welche in einer Dauerschleife den Philosophen zunächst nachdenken und dann mit dem linken und rechten Stick essen lässt. Die Verzögerung durch das Nachdenken und Essen können jeweils durch `snooze` implementiert werden. Beim Essen soll beachtet werden, dass synchronisiert auf die Sticks zugegriffen wird, das heißt, dass jeder Stick von maximal einem Philosophen gehalten werden kann.
- Implementieren Sie abschließend eine `main` Methode, die für eine Anzahl, die dem Programm als Parameter übergeben wird, zunächst die Stick-Objekte und ebenso viele Philosophen erzeugt und die Philosophen dann startet.

2. Überlegen Sie sich, unter welchen Umständen Deadlocks in Ihrer Implementierung auftreten und wie sie vermeiden werden können.

Aufgabe 3 - Synchronisation mit Semaphoren

1 Punkt

In der Vorlesung wurde das *Producer-Consumer-Problem* mit Hilfe von Semaphoren gelöst. Hierbei gingen wir von einem unbeschränkten Puffer aus.

```
1  Buffer buffer          = ...
2  Semaphore num          = 0;
3  Semaphore bufferAccess = 1;
4
5  // Producer
6  while (true) {
7    newproduct = produce();
8    P(bufferAccess);
9    push(newproduct, buffer);
10   V(bufferAccess);  // Reihenfolge
11   V(num);           // vertauschen
12 }
13
14 // Consumer
15 while (true) {
16   P(num);           // Reihenfolge
17   P(bufferAccess);  // vertauschen
18   prod = pull(buffer);
19   V(bufferAccess);
20   consume(prod);
21 }
```

1. In der Implementierung mit synchronisiertem Zugriff auf den Puffer führt der Producer zwei `V`-Operationen und der Consumer zwei `P`-Operationen hintereinander aus. Was geschieht, wenn man jeweils die Reihenfolge dieser Operationen vertauscht?
2. Die Synchronisation auf den Puffer ist nicht immer nötig. So können bei gewissen Implementierungen bestimmte Prozesse gleichzeitig auf einen Puffer zugreifen. Überlegen Sie, wie Sie die Synchronisation auf den Puffer verbessern können (Lockerung des wechselseitigen Ausschlusses).
3. Welche Änderungen müssen Sie vornehmen, wenn Sie einen beschränkten Puffer verwenden?