Lösungen zu den Pflichtaufgaben Concurrency - Cooperation

1. Concurrency 3 — Thread Synchronisation

1.1. Konto-Übertrag [PU]

Siehe Dokument *Lösungen zu den Übungsaufgaben Concurrency–Cooperation*.

1.2. Traffic Light [PU]

Siehe Dokument *Lösungen zu den Übungsaufgaben Concurrency–Cooperation*.

1.3. Producer-Consumer Problem [PU]

Siehe Dokument *Lösungen zu den Übungsaufgaben Concurrency–Cooperation*.

2. Concurrency 4 — Lock & Conditions, Deadlocks

2.1. Single-Lane Bridge [PU]

Siehe Dokument *Lösungen zu den Übungsaufgaben Concurrency–Cooperation*.

2.2. The Dining Philosophers [PA]

Beschreibung des Philosophen-Problems:

Fünf Philosophen sitzen an einem Tisch mit einer Schüssel, die immer genügend Spaghetti enthält. Ein Philosoph ist entweder am Denken oder am Essen. Um zu essen braucht er zwei Gabeln. Es hat aber nur fünf Gabeln. Ein Philosoph kann zum Essen nur die neben ihm liegenden Gabeln gebrauchen. Aus diesen Gründen muss ein Philosoph warten und hungern, solange einer seiner Nachbarn am Essen ist.

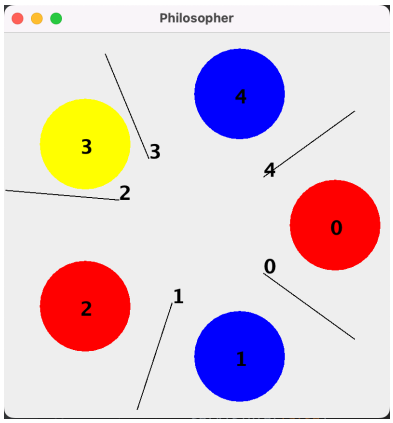

Figure 1. Philosopher UI

Das Bild zeigt die Ausgabe des Systems, das wir in dieser Aufgabe verwenden. Die blauen Kreise stellen denkende Philosophen dar, die gelben essende und die roten hungernde. Bitte beachten Sie, dass eine Gabel, die im Besitz eines Philosophen ist, zu dessen Teller hin verschoben dargestellt ist.

Die Anwendung besteht aus drei Dateien / Hauptklassen (jeweils mit zusätzlichen inneren Klassen):

`PhilosopherGui`

Ist das Hauptprogramm und repräsentiert das GUI (Java-Swing basiert). Die Klasse initialisiert die Umgebung `PhilosopherTable`, startet die Simulation und erzeugt die obige Ausgabe. Zudem werden Statusmeldungen der Philosophen auf der Konsole ausgegeben.

`PhilosopherTable`

Repräsentiert das Datenmodell. Sie initialisiert, startet und stoppt die Threads der Klasse `Philosopher`, welche das Verhalten und Zustände (THINKING, HUNGRY, EATING) der Philosophen abbildet und als innere Klasse umgesetzt ist.

`ForkManager`

Diese Klasse enthält die Strategie, wie die Philosophen die zwei Gabeln (Klasse `Fork`) aufnehmen (`aquireForks(int philosopherId)`) und zurücklegen (`releaseForks(int philosopherId)`).

a. Analysieren Sie die bestehende Lösung (vor allem `ForkManager`), die bekanntlich nicht Deadlock-frei ist. Kompilieren und starten Sie die Anwendung. Nach einiger Zeit geraten die Philosophen in eine Deadlock-Situation und verhungern. Überlegen Sie sich, wo im Code der Deadlock entsteht.

b. Passen Sie die bestehende Lösung so an, dass keine Deadlocks mehr möglich sind. Im Unterricht haben Sie mehrere Lösungsansätze kennengelernt.
In der umzusetzenden Lösung soll der `ForkManager` immer das Gabelpaar eines Philosophen in einer *atomaren* Operation belegen bzw. freigeben und nicht die einzelnen Gabeln sequentiell nacheinander. Dazu müssen beide Gabeln (links & rechts) auch verfügbar (`state == FREE`) sein, ansonsten muss man warten, bis beide verfügbar sind.

   - Es sind nur Anpassungen in der Datei `ForkManager.java` notwendig. Die `PhilosopherGui` und `PhilosopherTable`-Klassen müssen nicht angepasst werden.
   - Verändern Sie nicht das `public` interface des `ForkManager` – `acquireForks(int philosopherId)` und `releaseForks(int philosopherId)` müssen bestehen bleiben und verwendet werden.
   - Verwenden Sie für die Synchronisation Locks und Conditions!

Lösung 1a: `ch.zhaw.prog2.philosopher1a`

Der ForkManager wird so umgebaut, dass er "Gabelpaare" für den jeweiligen Philosophen verwaltet. Das `Lock` wird auf den `ForkManager` gesetzt, und jedes Gabelpaar verwendet eine eigene `Condition` (Wait-Set). Das Gabelpaar des aktuellen Philosophen wird nur akquiriert, wenn das Gabelpaar der benachbarten Philosophen (links & rechts) frei sind, ansonsten wird gewartet.
Da das Lock auf dem `ForkManager` ist, kann kein anderer Thread dazwischen grätschen. Dadurch ist jedoch auch die Parallelität reduziert. Wenn der Philosoph das Gabelpaar zurückgibt, wird der Status auf FREE gesetzt und jeweils via die Gabelpaare die wartenden Threads des linken und rechten Philosophen informiert (`signal` an jeweilige `Condition` des Gabelpaares).

Lösung 1b: `ch.zhaw.prog2.philosopher1b`

Gleich wie in Lösung 1a, wird das Lock auf den `ForkManager` gesetzt und jede Gabel verwendet eine eigene `Condition` (Wait-Set). Im Unterschied zu 1a stellt in der Methode `acquireForks` der `ForkManager` sicher, dass beide Gabeln gleichzeitig frei sind und nimmt diese nur dann, ansonsten wird auf die jeweilige Gabel in ihrem Wait-Set (Condition) gewartet. Auch hier ist die Parallelität reduziert, da das Lock auf dem `ForkManager` ist und kein anderer Thread gleichzeitig Gabeln akquirieren kann.
Trotz des gemeinsamen Locks könnten, ohne den Check auf freie Paare und das Warten auf die belegte Gabel, immer noch Deadlocks passieren, da mehrere Wait-Sets (Conditions) verwendet werden.
Wenn der Philosoph das Gabelpaar zurückgibt, werden die linke und rechte Gabel auf FREE gesetzt und die wartenden Threads informiert (`signal`).

Lösung 2: `ch.zhaw.prog2.philosopher2`

Alternative Lösung die zirkuläre Abhängigkeit vermieden, indem sich die Philosophen unterschiedlich verhalten. Philosophen mit ungerader Nummer sind Rechtshänder, mit geraden Nummern Linkshänder. Jeder Philosoph nimmt zuerst die Gabel seiner "stärkeren" Hand. Somit ist immer die erste Gabel "umstritten". Befindet sich ein Philosoph im Besitz dieser Gabel, ist die zweite immer frei.
Bei der Rückgabe er Gabel muss keine Reihenfolge eingehalten werden.

Da diese Lösung nicht die Anforderung an "atomares Nehmen der Gabelpaare" erfüllt ist es keine gültige Lösung, obwohl sie funktioniert und relativ einfach ist.