Christian-Albrechts-Universität zu Kiel
Institut für Informatik

Priv.-Doz. Dr. Frank Huch, Marius Rasch

Institut
für Informatik

1. Übung zur Vorlesung „Concurrent and Distributed Programming“

Abgabe am Monday, 15. April 2019 - 18:00

Aufgabe 1 - Dining Philosophers

1 Punkt

Implement the dining philosophers in Python. Realize the sticks as separate synchronization objects.

How can you awake sleeping philosophers? Is it possible with notify or do you need notify_all? Is it possible to put the wait call into an if statment or do you need a while loop?

In the lecture two strategies to avoid deadlocks were discussed. Implement them.

Alternatively, you can also create an implementation of the dying philosopher, which works only with synchronied without wait and notify. Is it also possible to avoid deadlocks in this implementation?

In the exercise session, both approaches are compared and discussed.

Aufgabe 2 - Counter

1 Punkt

1. Define a class `Counter`, which implements a continuous incrementing counter (starting with 0). The speed in which the counter increments should be set by a parameter of the constructor (in msec). This period the counter should wait until it increments the next time. Additionally the counter should get a name during its construction. Each time the counter increments it should write its name and its value to standard output.
2. Write a program which allows to start an arbitrary number of concurrent counters. To do so it should be possible to specify the speed of every counter when initializing the program. For instance three counter with different speed would be started by:

   ```
   python counter.py 300 500 1000
   ```

Aufgabe 3 - Extension of MVar

1 Punkt

In the lecture we stepwise developed an implementation of the class `MVar` to communicate using message passing.

Extend the implementation of `MVar` with `Lock` objects introduced in the lecture by the following methods:

- `read()` reads the `MVar` without making it empty. Blocks if `MVar` is empty.

Extend the implementation of `MVar` based on `Condition` objects last introduced in the lecture by the following methods:

- `read()` reads the `MVar` without making it empty. Blocks if `MVar` is empty.
- `try_put(v)` tries to put the value `v` but does not block; if the method fails it returns false.
- `try_take()` tries to take the value but does not block; if the `MVar` isn't empty it returns the value, otherwise it returns a value indicating that the `MVar` was empty. Therefore think about a reasonable type of the return value.
- `swap(v)` replaces the value (if any) and returns the old value; if the `MVar` is empty it blocks.
- Implement a variant of `take` or `put` which provides an additional timeout; after this timeout the method should stop blocking.

Hint: First of all think how to ensure that exactly one reader and one writer can reach certain points in corresponding methods. After this you need to synchronize reader and writer in a way, that the writer puts a value first and then notifies the reader (if any). The reader takes the value before other reader and writer gets notified.