HÁSKÓLI ÍSLANDS
Iðnaðarverkfræði-, vélaverkfræði- og tölvunarfræðideild

TÖL401G: Stýrikerfi / Operating Systems · Vormisseri 2022

Assignments 15–16 · To be solved until 15.3.2022, 13:00

Assignment 15

Last bolludagur, baker Jói was not able to produce enough *bollur*. To be prepared for next year, he wants to improve by using parallel baker processes that need to be synchronised via semaphores: one master baker process distributes work to three assistant baker processes. Each assistant baker endlessly produces one *bolla* after the other. For this, each assistant baker needs three ingredients: cream, chocolate, and a "naked" plain *bolla*. Each assistant baker has an infinite amount of only one type of ingredient that the two other assistant bakers in turn do not have. The master baker, however, has an infinite amount of all ingredients: he endlessly offers an arbitrary pair of two different ingredients to the assistant bakers and waits until one of the assistant bakers takes this pair: the assistant baker that has the matching third ingredient takes the offered pair of ingredients and assembles the final cream-filled and chocolate-topped *bolla* from it. Once the assistant baker has taken the offered pair of ingredients (but before starting assembling the ingredients), (s)he informs the master baker about this and the master baker offers a new random pair of two different ingredients to the assistant bakers.

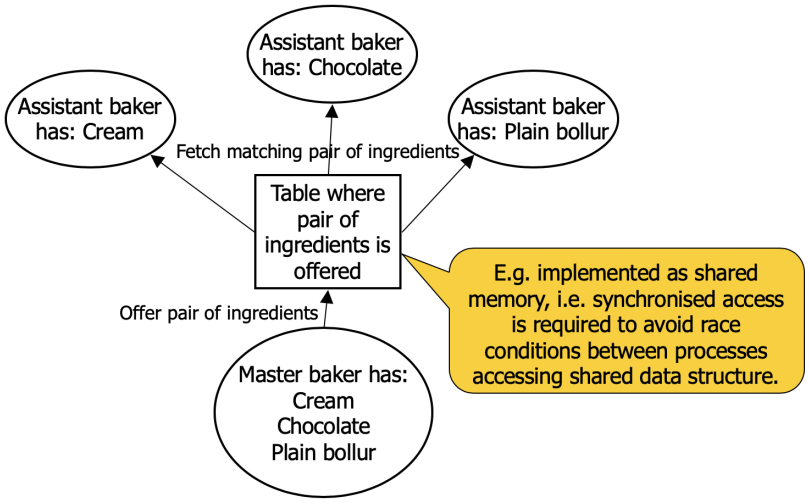

Figure 1: An image of Baker process

Use semaphores to create a pseudocode solution that guarantees that the distribution of ingredients is performed in a non-busy waiting and deadlock-free, but parallel synchronised way.[1] This means: only the assistant baker that has the matching third ingredient takes the offered pair of ingredients – the other two assistant bakers that do not have the matching

[1]For the actual offering and fetching of ingredients you can simply assume that there are functions `offer(ingredient1, ingredient2)` and `ingreditients:=fetch()` available as well as a function `assemble(ingredients)`. For determining the arbitrary ingredient pairs, you can use some `random` function.

third ingredient are sleeping until their pair of ingredients is offered, assembly itself may take place in parallel.

To this aim create commented pseudocode for:

- the initialisation of shared semaphores,
- for the master baker,
- for the three assistant bakers.

Use only the style of semaphores (and syntax) presented on the lecture slides!

Hint: it really helps to consider pairs of ingredients instead of individual ingredients.

Assignment 16

1. Show that monitors are at least as powerful as semaphores by implementing semaphores using the monitor concept: provide Java source code of a class that implements a weak counting semaphore (i.e. provide and implement Java methods `init`, `signal` and `wait` of a semaphore plus any internal variables that you need) using the monitor concepts provided by the Java language.

   Hint: think about what you need to implement the semaphore operations (among others: atomicity of operations, putting a calling thread to sleep and waking another thread up again) and which monitor concepts can be used to achieve this.

2. Use your semaphore implementation from part 1 to protect the critical section of the race condition problem from assignment 11! (Comparable to assignment 14, however instead of using the Java semaphores from package `java.util.concurrent.Semaphore`, use your own semaphore implementation from above.)
   Hint: two sample solutions for assignment 11 are available via Canvas as already described in assignment 14.

   Just like for assignments 11, 12, and 14, use as starting point the files from `tol401g_assignment16.zip` that you find in Canvas:

   - File `Assignment16.java` contains the main method that reads the number of iterations to be executed by *each* thread and prints the result out – **do not change that file**!
   - Use file `MyAssignment16.java` to add your implementation (it gets called by file `Assignment16.java`). Feel free to change that file except:
     (a) Do not change the name of this class (adding `extends` is OK) nor put it into another package.
     (b) Do not modify the name and input and output parameter of the main method.
   - You are allowed to add further classes.

   Run your program to assure that the printed value of `in` is correct even for large numbers of iterations!

3. Just as in Assignment 14, modify your solution from part 2. in a way that it becomes visible which thread is currently executing in its critical section: one thread should print a '`0`' when it enters its critical section, the other should print '`1`'.

Hints: By invoking method `setName(String)` *on a* `Thread`*, you can give it a name (such as* `"0"`*) and retrieve, e.g. inside the thread, the name via* `getName()`*.*
Use `System.err` *instead of* `System.out`*. (*`System.out` *is buffered and does not immediately print each character.* `System.err` *is not buffered and does therefore reflect the real order of how* `println` *is called by the different threads.)*

Just upload a zip file with the Java source code of part 3 (it includes parts 1 and 2).