HÁSKÓLI ÍSLANDS
Iðnaðarverkfræði-, vélaverkfræði- og tölvunarfræðideild

TÖL401G: Stýrikerfi / Operating Systems · Vormisseri 2022

Assignments 13–14 · To be solved until 8.3.2022, 13:00

Assignment 13

Use semaphores to solve the following synchronisation problem of a young family consisting of a mother, father, and two twin children (having the same behaviour): their day starts with each of them going to toilet first (`useToilet()`). The order of going to toilet is arbitrary and only one toilet exists. After the father has used the toilet, he prepares a drink for the children (`prepareDrinks()`); after the mother has used the toilet, she prepares food for the children (`prepareFood()`). If both food and drink are ready, the children have breakfast (`haveBreakfast()` – mother and father are so busy with the kids that they do not have breakfast themselves). Once the children have finished breakfast, the mother takes the child and drives the child to school (`takeAndDriveToSchool()`) and the father clears the table (`clearTable()`). Each process finishes behaviour after these steps.

Implement the approach described in the above text using pseudocode for the functions `mother()`, `father()`, `child()`, and the definition and initialisation of shared semaphores (`sem`). On the semaphores, only the methods `init(`*value*`)`, `wait`, and `signal` can be applied. Do not introduce any further variables.

Your solution needs to avoid busy waiting and to be obviously free of deadlocks and race conditions. The restriction of the concurrency of the processes needs to be minimised (e.g. do not require a certain order of using the toilet). You can use the following skeleton:

```
// Definition of shared semaphores
sem ...
sem ...
    ...
// Run 4 person family in parallel
parallel {child(), child(),
   mother(), father()}

child() {
     ...
     useToilet()
     ...
     haveBreakfast()
     ...
}
```

```
mother() {
     ...
     useToilet()
     ...
     prepareFood()
     ...
     takeAndDriveToSchool()
     ...
}
father() {
     ...
     useToilet()
     ...
     prepareDrinks()
     ...
     clearTable()
     ...
}
```

Assignment 14

1. Modify the solution of assignment 11 (or assignment 12 respectively) in a way that Java semaphores (see slide 6-75 for the API) are used to protect the critical section.

   In case, you have no working solution of assignment 11: two sample solutions for assignment 11 will be available via Canvas (Skjalageymsla→Verkefni): one that uses static variables and another that uses a shared object to share data between the two thread instances.

   Just like for assignment 11/12, use as starting point the files from `tol401g_assignment14.zip` that you find in Canvas:

   - File `Assignment14.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 `MyAssignment14.java` to add your implementation (it gets called by file `Assignment14.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!

2. Modify your solution from part 1. in a way that it becomes visible which thread is currently executing in its critical section: one thread should print to `System.err` 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()`.
   - Really 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.
   - Also the Gradescope tests get confused if you print to `System.out`.
   - Due to the parallel writing to `System.err`, the operating system has to synchronise the access to it, thus race conditions get extreme rare in part 2. Therefore, the Gradescope tests use millions of iterations which takes time...

3. Run your program and observe the pattern how the threads are executed on your machine; it might be interesting to see whether it is alternating `01010101`...or something else (such as `0000011111`...representing the time slices of a round-robin scheduler). **No need to answer and submit anything for this question!**

General comments concerning Java assignment submission

While you can create a PDF for the pseudocode solution of assignment 13, submit for assignment 14 only only the Java source code (`*.java`) to Gradescope.

1. Use **zip format** to create a single file archive of the multiple source code files! Use the same structure as in the `tol401g_assignment14.zip` files, i.e., no sub-directories, no Java package structure.

2. If Gradescope was able to compile the source from your uploaded zip file, it will run same basic sanity checks against your solution. **Failing already compilation or all or the majority of these checks means that something is wrong with your submission and you are likely to score 0 points** in the manual grading: Typically, compilation has failed because your zip file does not adhere to the structure described above. To fix these problems, you are can resubmit as often as you like.

3. You do not need to submit the Java source code for part 1 of assignment 14: just submit part 2 – it anyway includes the solution of part 1. You do not need to submit an answer for part 3 (just think about what the patterns tells you).

Preparation

Read chapter 6 up to slide 6-86, watch videos, play Deadlock Empire which will be about semaphores (pseudocode and Java). (Remaining sections of chapter 6 will be covered within next week.)