Concurrency — Exercise 4
Rust I

Prof. Dr. Oliver Haase

In this exercise, you will write a **serial** solution of example 3, i.e. the factorizer service that reads and writes its simple cache, in Rust. We'll break down the problem into smaller steps.

Problem 1

In this problem, you will write a **serial** solution of example 3, i.e. the factorizer service that reads and writes its simple cache, in Rust. To break down the problem into smaller steps, start with a factorizer service *without cache*. Your approach should adhere to the following description:

- Use a fixed size array for the storage and exchange of the calculated factors. Do not use a Rust vector, or a similar dynamic collection type. The reason is two-folded: (1) We have not yet covered structs, and (2) using an output parameter is a good way to practise reference borrowing. For the length of the array, use a reasonable value, and then simply assume that all factors will fit into the array.
- Declare and define a result array inside the top level `service` function and pass this array as an output parameter into the `factorizer` method. The `service` function gets as input the number to be factorized and returns as output the resulting array of factors. Unused array components should contain 0.

  As mentioned above, in the step the `service` function does not use a cache.
- The `factorizer` method computes the factors and fills them into the output array parameter.
- Write a `print_result` function that gets as input a number and a factors array, and that prints output of the form:

  ```
  32568 = 2 * 2 * 2 * 3 * 23 * 59
  ```
- Write a `main` function that calls the service function multiple times for different numbers to factorize, and that prints the results on the screen.

Problem 2

Now, extend your solution with a cache that's read and written by the `service` function. Follow these recommendations:

- On the top level, i.e. outside any function, define two static variables for the cache like this:

  ```
  static mut LAST_NUMBER: u64 = 0;
  static mut LAST_FACTORS: [u64; MAX] = [0; MAX];
  ```

- In the `service` function, access to these static variables can only happen inside unsafe blocks. An unsafe block is a code block labelled with the `unsafe` keyword, like this:

  ```
  unsafe {
      // code block containing, e.g., access to static variables
  }
  ```

- Within the `service` function, print on the screen whether you've run into a cache hit or a cache miss.

- In your `main` function, run the `service` function several times including two subsequent calls with the same number. The produced output should look something like this:

  ```
  cache miss
  32567 = 29 * 1123

  cache miss
  32568 = 2 * 2 * 2 * 3 * 23 * 59

  cache hit
  32568 = 2 * 2 * 2 * 3 * 23 * 59
  ```

Have fun and good luck!