Homework Assignment 1 (12.5 points)

4DC3, Winter 2019

Due: February 6, 23:59pm

Marking

This assignment accounts for 12.5% of your final mark.

Instructions

Please submit your solution on Avenue as a single ZIP before the deadline. Only the last submitted version will be considered for grading. You are encouraged to use the `mix` tool to create a project directory (see the tutorial slides). Please see the course outline for the policy on late submissions.

Problem Set

Problem 1. Write a server that computes Fibonacci numbers, defined as follows:

$$F(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ F(n-1) + F(n-2) & \text{if } n > 2 \end{cases}$$

Place the following code into a module `Fib`.

- Write a function `fib_calc(n)` that computes $F(n)$ by directly following the above definition.
- Add a function `server(caller)` to the module. The *server process* is started by spawning `server(caller)`, where caller is the PID of the calling (i.e. parent) process:
  - The server should send a message {`:ready,server_pid`} to the parent process, indicating that it is ready to receive a new task.
  - What the server does next depends on the received message. The server should handle 2 types of messages:
    1. If the server receives {`:compute, n, client`}, it computes $F(n)$ and sends the result to the process with pid `client` by using the message format {`:answer, n, result`}, where variable `result` $= F(n)$. Subsequently, the server recursively calls the function `server`.
    2. If the server receives a message {`:shutdown`}, it terminates using the function `exit(:normal)`.

[Points: 3]

Problem 2. Extend the design of Problem 1 by adding a module `Scheduler`. (For the purpose of this assignment, you can place the `Fib` module and this module into the same source file.)

- Add a function `start(num_servers,job_list)`, where `num_servers` is an integer and `job_list` is a list of integers: We call the process executing this function the *scheduler*. The `start` function is used by the scheduler to do some initial setup: First, the scheduler spawns `num_servers` many instances of the Fibonacci server process implemented in Problem 1 and then calls the function `run` (see below) as follows: `run(num_servers, job_list, [])`.
- Add the function `run(num_servers, job_list, result_list)`: The scheduler will recursively call this function to (iteratively) empty its `job_list`.
  - The scheduler listens for messages from the server instances. Upon receiving a message {`:ready,server_pid`} from some server process with PID `server_pid`, the scheduler checks if its (current) `job_list` is nonempty and, if so, sends a message {`:compute, n, self()`} to the server, where n is the head of `job_list`. Otherwise, if the job list is empty, the scheduler sends a {`:shutdown`} message to the server.
  - Upon receiving {`:answer, n, result`} from a server, the scheduler keeps track of the computed value in `result` by adding it to `result_list`.
  - The scheduler process terminates and returns `result_list` once it has shutdown all server instances. (Note that this implies that it has received an answer for each value in its original `job_list`.)

[Points: 5]

Problem 3. Suppose we start the scheduler by calling `Scheduler.start(2,[5,8])`. Draw a simple space-time diagram (see lecture slides) that shows how the 3 processes (scheduler and two server instances) communicate. Your diagram should depict all messages as described in Problems 1 and 2. Note that there is no unique space-time diagram since we assume an *asynchronous* message passing system. You do not need to annotate specific compute/deliver events; just drawing the message pattern is sufficient.

[Points: 2]

Problem 4. The direct implementation of the Fibonacci numbers as described in Problem 1 is not very efficient. In particular, it seems that the recursive definition of $F(n)$ causes previously computed values of $F(n)$ to be recomputed again and again. Implement a more efficient version of function `fib_calc` that avoids this issue by reusing already computed values.

Hint: `fib_calc` will need to call a helper function that does the actual work and which keeps an updated "cache" (e.g. implemented using a Map) of already computed Fibonacci numbers.

[Points: 2.5]