Multicore Programming

Lab 4: Message Processing and Distribution

Assistant: Janwillem Swalens
Mail: jswalens@vub.ac.be
Office: 10F719

5 March 2014

Send your solutions to `jswalens@vub.ac.be` at the latest on Monday, at 23:59.

1 Exercise 1: Message Receive Order

`priority.erl`

Sometimes, certain messages need to be processed with an increased priority, for instance to manage high load or gather debug information. `priority.erl` provides you with a simple skeleton and a test case. The function `priority/1` should keep a history containing the numbers of the messages it has received. Adapt it to receive and process the high priority messages before the ones with low priority. Make sure that, even when the message buffer is not empty, high priority messages are processed first.

After completing the task, try to explain the order in which messages are received in the `receive` statement and in which order the matching is applied.

2 Exercise 2: Actors as Concurrent Data Structures

`my_queue.erl`

The only way to represent shared data in a pure actor system is to represent it as an actor. Certain data structures can utilize the underlying system nicely.

Implement a queue actor that can receive the messages:

- `is_empty`
- `{enqueue, Value}`, and
- `dequeue`.

Reuse the existing client interface given in `my_queue.erl`. The file provides two simple tests, and expects a function `queue_empty/0` to be implemented.

Do not use any data structures to represent the queue, i.e., do not use lists, maps, sets, or similar things. Instead, find a way to encode the queue using the basic elements of Erlang's actor system.

After completing the task, reason about the resulting implementation. How did you realize the queue? What happens when you dequeue an empty queue?

3 Exercise 3: Distributed Erlang, from ping pong to chat

`pingpong.erl`

3.1 Distributed Erlang

Setup two local instances of Erlang to communicate with each other. Adapt the ping pong program to work in this distributed environment. Make sure the pings and pongs are printed out on the expected Erlang instance.

You can find documentation on distributed Erlang here:

- http://www.erlang.org/doc/reference_manual/distributed.html
  http://www.erlang.org/doc/getting_started/conc_prog.html#id67453
  Official documentation on distributed Erlang.
- http://www.erlang.org/download/erlang-book-part1.pdf
  See Chapter 6 on *Distributed Programming* from the book *Concurrent Programming in Erlang* by Joe Armstrong et al.
- http://learnyousomeerlang.com/distribunomicon#setting-up-an-erlang-cluster
  Section *Setting up an Erlang cluster* from the book *Learn You Some Erlang for Great Good!* by Fred Hebert.

3.2 Chat[1]

Afterwards, extend the ping pong to a simple chat:

1. Enable the two fixed instances to communicate via user input. Print out the input of the first Erlang instance in the terminal of the second instance, and vice versa. This is a simple chat application.
2. Extend the chat with a master actor that works as an intermediary, broadcasting the messages to all connect clients.

[1] Optional