Erlang: philosophers and concurrency

Johan Montelius

November 27, 2013

Introduction

In this assignment you should implement the behavior of five philosophers that are sitting at a round dining table. The trick is to allow each philosopher to get something to eat, a task that does not sound to be very difficult.

The problem is that the five philosophers are sitting at the dinner table with a bowl of noodles in front of them, and a chopstick between each of them. We thus have five philosophers and only five chopsticks, this is the problem.

When a philosopher decides to eat (they sit and think most of the time), she will pick up the two chopsticks next to her and eat from the bowl of noodles. When she is done, she will simply return the chopsticks to their places. If the philosophers do not eat that often things will probably work out just fine, the problem occurs when several philosophers decide to eat at the same time.

You should complete a simple program that implements the behavior of the philosophers and run some experiments to see when things go wrong. You should also provide a solution to the problem that at least allow some of the philosophers to get some food.

1 A chopstick

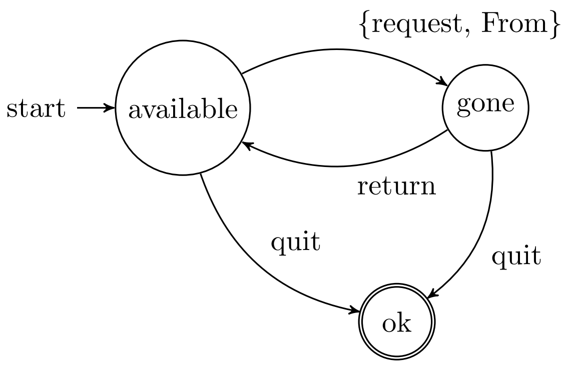
The location of a chopstick is represented by a process. The state of the process is either:

- `available` : if a chopstick is present or,
- `gone` : if the chopstick is taken

A location will start in the state available, and then wait for a *request message*. When the location process receives this message it should return a *granted* message and move to the state *gone*. In this state the location process will accept a *return message* and can then return to the available state. Messages that are not currently handled remain in the message queu.

In both states the process should be able to receive a *quit message* in which it will simply terminate.

Implement this process in a module `chopstick` and export a function `start/0` that spawns the process and returns the process id. When you



spawn the process use `spawn_link/1` to make sure that the chopstick process dies if the mother process dies (and vice verse). This is some skeleton code to give you the structure of the implementation.

```
start() ->
    spawn_link(fun() -> ... end).

available() ->
    receive
        ... ->
             :
        quit ->
            ok
    end.

gone() ->
    receive
        ... ->
            :
        quit ->
            ok
    end.
```

We should try to keep the internals of the location process as hidden as possible from the users of the module. We therefore provide a functional interface so that a user of the module does not need to know the structure of all messages.

```
request(Stick) ->
  Stick ! ...,
  receive
    ... ->
        ok
  end.
```

Provide similar functions for returning the stick and terminating the process. We will change things later so you will see that it is very nice to only allow the philosophers to use the functional interface.

2 A philosopher

A philosopher is either dreaming (some call it thinking), waiting for a chopstick or eating. In the dreaming state the philosopher does nothing until she decides that it is time to eat. She will then request her left chopstick and her right chopstick, if everything works she can start to eat. A philosopher will eat for while and then return the chopstics.

You can implement the dreaming and eating time by using the library function `timer:sleep/1` that will simply wait for a number of milliseconds before continuing. If you want to have some randomness you can use the library function `random:uniform/1`. This sequence will make a process sleep for a random time.

```
sleep(T,D)
    timer:sleep(T + random:uniform(D)).
```

Implement the philosopher in a module called `philosopher` and provide a function `start/5` that spawns a philosopher process (use spawn_link/1). The procedure should take the following arguments.

- `Hungry`: the number of times the Philosopher should eat before it sends a `done` message to the controller process.
- `Right` and `Left`: the process identifiers of the two chopsticks.
- `Name`: a string that is the name of the philosopher, used for nice logging.
- `Ctrl`: a controller process that should be informed when the philosopher is done.

Add some nice logging information to your process so that you can track what is happening. A philosopher could for example print a message when it receives a chopstick:

```
io:format("~s received a chopstick~n", [Name])
```

The `~s` sequence will print a string, given as an element in the list given as the second argument.

3 Dinner at the table

If you have the two modules working we can seat the philosophers around the table. We first create the locations and then start the philosophers. In a module called `dinner`, define the following function:

```
start() ->
   spawn(fun() -> init() end).

init() ->
   C1 = chopstick:start(),
   C2 = chopstick:start(),
   C3 = chopstick:start(),
   C4 = chopstick:start(),
   C5 = chopstick:start(),
   Ctrl = self(),
   philosopher:start(5, C1, C2, "Arendt", Ctrl),
   philosopher:start(5, C2, C3, "Hypatia", Ctrl),
   philosopher:start(5, C3, C4, "Simone", Ctrl),
   philosopher:start(5, C4, C5, "Elizabeth", Ctrl),
   philosopher:start(5, C5, C1, "Ayn", Ctrl),
   wait(5, [C1, C2, C3, C4, C5]).
```

We're starting all processes under a controlling process that will keep track of all the philosophers and also make sure that the chopstick processes are terminated when we're done.

```
wait(0, Chopsticks) ->
    lists:foreach(fun(C) -> chopstick:quit(C) end, Chopsticks);
wait(N, Chopsticks) ->
    receive
        done ->
            wait(N-1, Chopsticks);
        abort ->
            exit(abort)
    end.
```

If things go wrong and a process terminates with an error it will kill all linked processes. If things are stuck in a dead-lock we can send an `abort` message to the controller process that then will exit with an error and kill all other processes.

Now it's time to see if the philosophers will be able to dream and eat.

4 Experiments

Experiment with the dinner, will the philosophers always be able to eat? What happens if you decrease the time it dreams? What happens if you introduce an artificial delay between the receiving of the first chopstick and requesting the second?

4.1 break the dead-lock

To break out of a potential dead-lock situation we can change the request function in the chopstick module. Let's pass a second argument to the function that specifies how many millisecond we are willing to wait for a chopstick.

```
request(Stick, Timeout) ->
   Stick ! ...,
   receive
      ... ->
        ok
   after ... ->
        no
  end.
```

Change your implementation of the philosphers to use the new interface. At this time it is worth understanding a bit about the random module. It will as you have figured out generate a random number each time we can uniform/1 but, the sequence is the same each time we create a new process. A process will use the same sequence every time it is created. This is very nice when we debug programs since we then eliminate one source of indeterminism, but not as fun when we want processes to behave different each time we run it. Do some reading and provide each philosopher with a unique seed value, if you do it right you can have different execution patterns at will.

So you have broken the dead-lock, or so you think, but what is actually happening? What happens when a philosopher gives up? You have to do some thinking but the solution is quite simple once you trace what is happening.

4.2 asynchronous requests

The solution that you have now is quite boring in that a philosopher will first request the left chopstick and only when this is delivered will it try to grab the right chopstick. How about sending a request to both chopsticks first and then wait for the replies? Change the request function and the provide a *granted* function that does the waiting.

If a philosopher gives up, how do we keep track of which chopsticks that was actually obtained? Is this a tricky problem or a non-problem?

4.3 a waiter

Can you provide a better strategy for the philosophers so that they can eat and dream without ending up in a dead-lock? What happens if you provide a waiter that controls how many philosophers that can eat at any given time. How would this help the situation? How many philosophers can try to eat without ending up in a dead-lock? How smart does the waiter need to be?

4.4 benchmark

Run some benchmarks and try to figure out how long time it takes for a set of philosophers to eat a given number of times. Use the algorithms that do risk ending up in dead-locks and try to be as aggressive as possible.

Can you work with a increasing "back-off" time so that a philosopher will wait for a while before trying to grab the chopsticks if it has failed once? Can the system adapt itself so that it runs smoothly without too many failed attempts? Is there a trade-off between being aggressive and over-all throughput?

4.5 avoid the dead-lock

Is there a small change in the system that will avoid ever landing in a dead-lock situation? Can you guarantee that all philosophers will eventually get to eat? Is the system fair?