Erlang: philosophers and concurrency

Johan Montelius

October 30, 2012

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 a dinner table with a bowl of noodles in from of them and with a chop stick between each of them. When a philosopher decides to eat (they sit and think most of the time), she will pick up the two chops ticks next to her and eat from the bowl of noddles. When she is done, she will simply return the chop sticks 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 chop stick

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

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

A location will start in the state `available` and then wait for a message:

- {`request, From`}

This is a message from a philosopher, with process id `From`, that requests the chop stick. When the location process receives this message it should return the message:

- `granted`

and move to the state `gone`. In this state the location process will accept the message

- `returned`

It then moves into the state `available`. It is then prepared to accept another request. In both states the process should be able to receive the message `quit` 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_1ink/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).

init() ->
    ...

available() ->
    receive
        ... ->
             :

        quit ->
            ok
    end.

gone() ->
    receive
        ... ->
            :

        quit ->
            ok
    end.
```

2 a philosopher

A philosopher is either dreaming (some call it thinking), waiting or eating. In the dreaming state the philosopher does nothing until it decides that it is time to eat. It will then send messages to both of its adjacent locations to request the chopsticks and then enter the waiting state.

In the waiting state the philosopher will wait for two `granted` messages and then enter the `eating` state. It will eat for a while and then return the

chopsticks and continue to dream.

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 between 500 and 1000 ms.

```
Time = 500+random:uniform(500),
timer:sleep(Time),
```

Implement the philosopher in a module called `philosopher` and provide a function `start/5` that spawns a philosopher process. 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 received 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 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, "Confucios", Ctrl),
    philosopher:start(5, C2, C3, "Avicenna", Ctrl),
    philosopher:start(5, C3, C4, "Plato", Ctrl),
    philosopher:start(5, C4, C5, "Kant", Ctrl),
    philosopher:start(5, C5, C1, "Descartes", 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) -> C ! quit end, Chopsticks);
wait(N, Chopsticks) ->
    receive
        done ->
            wait(N-1, Chopsticks);
        abort ->
            erlang: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?

You can set a time limit on for how long you're willing to wait for a message by using the `after` construct.

```
    receive
        {granted, From} ->
                  :
                  :
```

```
    after 5000 ->
             io:format("~s giving up~n", [Name]),
              :
end.
```

Note - when you give up you might have a chopstick in your hand that should be returned. You also have at least one granted message that might arrive later, how do you handle this?

Try to use this to avoid deadlocks; how is this related to liveness and freedom from starvation?

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 it self so that it runs smoothly without too many failed attempts?

Can you provide a better strategy for the philosophers so that they can eat and dream without ending up in a dead-lock? Is this a solution that can be applied in general?

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 and how many philosophers can try to eat without ending up in a dead-lock?