Erlang assignment 3

Parallel and distributed programming

Isak Samsten

Spring, 2021

Introduction

Advanced assignments for parallel Erlang using `gen_server`. This week has three assignments to test your knowledge of concurrency and Erlang behaviors. The tasks are awarded a maximum of 20 points. The grades are distributed according to Table 1.

Table 1: Point to grade conversion table

Point	Grade
$0 - 8$	F
$9 - 10$	E
$11 - 12$	D
$13 - 15$	C
$16 - 17$	B
$18 - 20$	A

Submission

The following files should be submitted to iLearn:

Problem 1: `monitor.erl` (using `supervisor`)

Problem 2: `bank.erl` (using `gen_server`)

Problem 3: `ring.erl`

Problem 4: `gen_produceconsume.erl`

Problems

Problem 1 (5 points)

Implement the `monitor`-module from Assignment 2 `Problem 1` using the `supervisor` behavior. The functionality should be exactly the same, i.e., the monitor module starts the `double` process and monitors it, restarting it if it crashes. **Note that only `monitor.erl` should be submitted!**

Problem 2 (5 points)

Implement the `bank`-module from Assignments 2 *Problem 1* using the *gen_server* behavior. The inferface should exactly correspond to that of Assignment 2. Note that you can reuse much of your application logic.

Problem 3 (5 points)

Implement an Erlang module `ring`, with a function `start(N, M)` which should create a ring of `N` processes, then send an integer `M` times round the ring. The message should start as the integer `0` and each process in the ring should increment the integer by `1`. After the message has been round the ring `M` times its final value should be `N * M`. `ring:start(N, M)` should return the value of the last message (which should be `N * M`) and make sure that all the processes in the ring have terminated.

Problem 3 (5 points)

Implement a generic producer consumer behavior module that can be used to implement the produce/consume pattern with custom tasks. The `gen_produceconsume`-behavior defines `handle_produce/1` and `handle_consume/1` callback[1]. `handle_produce/1` takes an input and returns {`ok, Task`}, where `Task` is a task to be consumed by `handle_consume/1` and `handle_consume/1` returns `ok`. The module exports the following functions:

Function	Description
`start(Callback, T)`	Initialize the produce consume buffer with `Callback` which allows a maximum of `T` tasks in the queu. Return the process identifier of the buffer.
`stop(Pid)`	Stop `Pid`.
`produce(Pid, T)`	Add T to buffer and call `Callback:handle_produce(T)` which puts the result in the queue for processing.
`consume(Pid)`	The buffer consumes the next work in the queue using `Callback:handle_consume/1`

More specifically, `gen_produceconsume` encapsulate the behavior of the bounded buffer problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the queue, and start again. The consumer is consuming the data, one piece at a time. The problem is to make sure that the producer won't try to add data into the queue if it's full and that the consumer won't try to remove data from an empty queue. Here, `handle_produce/1` and `handle_consume/1` are user specified. Both `produce/2` and `consume/1` can be called concurrently. `produce/2` blocks if the queue is full, and `consume/1` blocks if the queue is empty. In Example 4.1, we allow a maximum of 3 items in the buffer at the same time (see `start(?MODULE, 3)`).

[1] Implementors should receive a warning if the callback is not exported, i.e., you should use `-callback` to define your callback function.

Example 4.1

```
-module(test).
-behaviour(gen_produceconsume).
-export([handle_produce/0, handle_consume/1]).

handle_produce(N) ->
    {task, N}.

handle_consume({task, N}) ->
    io:format("Consuming: ~p~n", [N]),
    ok.

test() ->
    P = gen_produceconsume:start(?MODULE, 3),
    spawn(fun () -> lists:foreach(
                        fun (I) ->
                                gen_produceconsume:produce(P, I)
                        end, lists:seq(1, 10))
          end),
    spawn(fun () -> lists:foreach(
                        fun (_) ->
                                gen_produceconsume:consume(P)
                        end, lists:seq(1, 10))
          end)
    ok.
```