Rock, Paper, Scissors

By Ken Friis Larsen (`kflarsen@diku.dk`) with minor adjustments by Oleks Shturmov (`oleks@oleks.info`) and Mikkel Kragh Mathiesen (`mkm@di.ku.dk`)

Last revision: October 12, 2022

Part 1: Warm up

In the warm-up section of assignment 4 you were asked to implement an Erlang module called `async`, exporting the function `new/2`, `wait/1`, and `poll/1`. In this task, you should implement the same 3 functions, and 2 more, but now, using OTP. We repeat the descriptions of the initial 3 functions below, for reference:

- `new(Fun, Arg)` starts a concurrent computation that computes `Fun(Arg)`. It returns an action ID.
- `wait(Aid)` waits for an asynchronous action to complete, and return the value of the computation. If the asynchronous action threw an exception, then the exception is re-thrown by `wait`.
- `poll(Aid)` checks whether an asynchronous action has completed yet. If it has not completed yet, then the result is `nothing`; otherwise the result is either `{exception, Ex}` if the asynchronous action raised the exception `Ex`, or `{ok, Res}` if it returned the value `Res`.
- `wait_catch(Aid)` that waits for an asynchronous action to complete, and return either `{exception, Ex}` if the asynchronous action raised the exception `Ex`, or `{ok, Res}` if it returned the value `Res`.
- `wait_any(Aids)` that waits for any of the supplied asynchronous actions to complete, where `Aids` is a non-empty list of asynchronous actions. If the first to complete throws an exception, then that exception is re-thrown by `wait_any`. Returns a pair `{Aid, Res}` where `Aid` is the action ID that completed with the result `Res`.

Before you start you should take a moment to consider how many processes you need and what role(s) each process have. How the processes should communicate (which protocol). Which data each process is in charge of. Then you should consider whether `gen_server` or `gen_statem` is best for each process (we don't expect you to use other parts of OTP).

Part 2: Rock, Paper, Scissors

Objective

The learning objectives of this assignment are:

- Gain hands-on programming experience with OTP.
- Practise using callback modules
- Orchestrating processes with a somewhat complicated communication protocol.

Terminology

This assignment is about making a game server for the rock–paper–scissors game. The traditional way to play a *round* of the game is that two people face each other, and then each player makes a *move* by forming their hand in the shape of either a *rock* (a clenched fist), a piece of *paper* (a flat palm), or a pair of *scissors* (index and middle fingers extended). The winner is then determined by the rule 'paper wraps rock, rock blunts scissors, and scissors cut paper'. If both players form the same object, then the game is a tie and neither wins. A *full game* is often a best-of-N rounds, where the game consist of a number of *rounds* until one of the players has achieved a more than $N/2$ number of *wins* (N is agreed on in advance). Note that a full game may consists of a unbounded number of rounds.

However, the traditional way to play the game often results in long disputes about how the players should make their moves simultaneous and whether they did so. Thus, we need a game server to avoid such disputes. Because we anticipate that our game server will be hugely popular we want it to be able to handle many concurrent games.

The game server consists of a *game broker* and a number of *game coordinators*. The broker takes care of matching up players, and then assign a (perhaps newly started) coordinator when two players have been matched up. The coordinator orchestrates a game (consisting of a number of rounds) between two players.

Two players match up if they want to play a best-of-N game with the same N.

The `rps` module

Implement an Erlang module `rps` with the following API:

- `start()` for starting a broker.

  Returns `{ok, BrokerRef}` on success or `{error, Reason}` if some error occurred.

- `queue_up(BrokerRef, Name, Rounds)` for queueing up for a best-of-`Rounds` game. `Name` is the name of the player and can be any Erlang term and `Rounds` should be a non-negative integer.

  Returns `{ok, OtherPlayer, Coordinator}` on success, where `OtherPlayer` is the name of the other player and `Coordinator` is a reference to a game coordinator; `server_stopping` if the server is being drained (see the description for `drain`); or `{error, Reason}` if some other error occurred.

- `move(Coordinator, Choice)` for making a move in a game, where `Coordinator` is a reference to a game coordinator and `Choice` is one of the atoms `rock`, `paper`, or `scissors`. Returns either:

  - `tie` if the round was a tie. This include the case where both players play an invalid choice (e.g., `laser`).
  - `win` if the player won the round.
  - `{loss, WinningChoice}` if the player lost the round, where the winner played `WinningChoice`. If the player plays an invalid choice (and the opponent plays a valid choice), this counts as a loss for the player.
  - `{game_over, Your, Other}` if the game is over and you won `Your` rounds and the other player won `Other` rounds.
  - `server_stopping` if the server is being drained (see the description for `drain`).

  Only players (processes) that have been assigned to the coordinator by a broker should call this function, likewise after this function has returned `server_stopping` or `{game_over, Your, Other}`, it is no longer valid for that player to call the function. If these requirements are not fulfilled it is unspecified what the function will return or do (it might even block forever).

- `statistics(BrokerRef)` for getting some statistics about the server. Returns a tuple with four elements `{ok, LongestGame, InQueue, Ongoing}` where `LongestGame` is the longest game (number of rounds, including ties) completed on the server so far, `InQueue` is the number of players, currently waiting to be matched up, and `Ongoing` is the number of games currently going on.

- `drain(BrokerRef, Pid, Msg)` for stopping the broker and all coordinators. Players queued up and new players should be sent an error response to their `queue_up` calls, and players in an ongoing game should get a `server_stopping` response to their `move` calls.

The function is non-blocking, hence it might return before the draining is completed. After the draining is completed `Msg` is sent to `Pid`, unless `Pid` is the atom `none`. The sending of `Msg` to `Pid` should be as a plain Erlang message (that is, by using `!`).

Using the `rps` module

The following example shows a computer player that follows the simple strategy of always making the `rock` move.

```
-module(rock_bot).
-export([queue_up_and_play/1]).

queue_up_and_play(Broker) ->
    {ok, _Other, Coor} = rps:queue_up(Broker, "Rock
        bot(tom)", 3),
    rock_to_game_over(Coor).

rock_to_game_over(Coor) ->
    case rps:move(Coor, rock) of
        {game_over, Me, SomeLoser} ->
            {ok, Me, SomeLoser};
        server_stopping ->
            server_stopping;
        _ -> rock_to_game_over(Coor)
    end.
```

Testing `rps`

You tests should be in a module called `test_rps` in the `tests` directory, this module should export at least one function, `test_all/0`. We will run your tests against our special `rps` module(s), and OnlineTA might not do any testing of your code unless it is minimally satisfied with your tests.

Note that since we'll run your tests against *our* `rps` module, the tests that are performed from `test_all/0` shouldn't rely on things specific to *your* solution. Thus, if you have tests that are specific to your implementation, then you might want to export a `test_everything/0` function as well (that could call `test_all/0`).

Hand-in Instructions

You should hand in two things:

1. A short report, `report.pdf` (normally 1–3 pages, excluding figures), for the main (not warm-up) part only, explaining the main design of your code, and **an assessment** of your implementation, including what this assessment is based on. See the generic instructions regarding the report in previous assignments.

   In your report you should describe what state your server (and other relevant processes) maintain. Likewise, consider making a diagram of your processes. How do the processes interact, and how many (kinds of) processes are there in the system?

   You should consider using `gen_statem` for implementing the coordinators. In your report, you should discuss whether you have used `gen_statem` or not, and why, or why not.

2. A ZIP archive `code.zip`, mirroring the structure of the handout, containing your source code and tests.

Make sure that you adhere to the types of the API, and test that you do.

To keep your TA happy, follow these simple rules:

1. The Erlang compiler, with the parameter `-Wall`, should not yield any errors or warnings for your code.
2. You should comment your code properly, especially if you doubt its correctness, or if you think there is a more elegant way to write a certain piece of code. A good comment should explain your ideas and provide insight.
3. Adhere to the restrictions set in the assignment text, and make sure that you export all of the requested API (even if some functions only have a dummy implementation).
4. Describe clearly what parts of the assignment you have solved.

End of assignment text.